CLEVERSON CARNEIRO TREVENZOLI

UM MECANISMO PARA EXTENSIBILIDADE DE ASPECTJ EM TEMPO DE COSTURA

Dissertação apresentada a Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

VIÇOSA
MINAS GERAIS - BRASIL
2013

Ficha catalográfica preparada pela Seção de Catalogação e Classificação da Biblioteca Central da UFV

T

T812m
2013

Trevenzoli, Cleverson Carneiro, 1988-
Um mecanismo para extensibilidade de AspectJ em tempo de costura / Cleverson Carneiro Trevenzoli. – Viçosa, MG, 2013.
xi, 96f. : il. ; 29cm.

Inclui apêndices.
Orientador: Vladimir Oliveira Di Iorio.
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f. 55-57.

1. AspectJ (Linguagem de programação de computadores).
2. Java (Linguagem de programação de computadores).
3. Métodos orientados a aspectos (Computação).
I. Universidade Federal de Viçosa. Departamento de Ciência da Computação. Programa de Pós-Graduação em Ciência da Computação. II. Título.

CDD 22. ed. 005.133

CLEVERSON CARNEIRO TREVENZOLI

UM MECANISMO PARA EXTENSIBILIDADE DE ASPECTJ EM TEMPO DE COSTURA

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Camputação, para obtenção do título de *Magister Scientiae*.

APROVADA: 26 de abril de 2013

Alcione de Paiva Oliveira
(Coorientador)

Leacir Nogueira Bastos

Vladimir Oliveira Di Iorio
(Orientador)

Aos meus pais, Cláudio e Maria das Graças,
que foram e ainda são meus educadores primordiais,
responsáveis pela minha vida e pela formação do meu caráter...

“Nobody is so wise that have nothing to learn,
not so foolish as to have nothing to teach.”
(Blaise Pascal)

AGRADECIMENTOS

A Deus, pelo dia após dia, pela minha vida, pela oportunidade e pelas batalhas diárias que se fizeram necessárias para o aprendizado e crescimento no fim.
Aos meus pais, Cláudio e Maria das Graças, pelos bons ensinamentos, pelo amor, carinho e, principalmente, por confiarem em mim.
A toda minha família, por estarem presentes em minha vida.
A Universidade Federal de Viçosa e ao Departamento de Informática, pela oportunidade de fazer o curso de Ciência da Computação e, posteriormente, o mestrado.
Ao professor Vladimir, por ser um exemplo para mim além de um grande orientador, pela atenção, paciência, motivação e ajuda, sempre que necessário.
Aos meus amigos, pela motivação, pelos bons momentos juntos e por estarem sempre presentes nas maiores dificuldades.
Ao Projeto Reuni, pela concessão da bolsa de mestrado.
A todos que, diretamente ou indiretamente, deram sua contribuição para o desenvolvimento deste trabalho.

Sumário

Lista de Figuras **vi**

Lista de Tabelas **viii**

Resumo **ix**

Abstract **x**

1 Introdução **1**

1.1 O Problema e a sua Importância 2
1.2 Motivação 3
1.3 Objetivo 4
1.4 Como o Texto está Organizado 4

2 Referencial Teórico **5**

2.1 Programação Orientada a Aspectos 5
2.1.1 AspectJ e suas Limitações 8
2.2 O Compilador AspectBench 8
2.2.1 Polyglot 9
2.2.2 Soot 11
2.3 A extensão *eaj* 14
2.4 Resíduos e Código Residual 14
2.5 A Linguagem XAJ 15
2.6 Trabalhos Relacionados 18

3 Extensibilidade de Pointcuts em XAJ **21**

3.1 Exemplo para Motivação 21
3.2 Solução em XAJ para o Problema Apresentado 24

4 Detalhes de Implementação **29**

4.1 Implementação de Mecanismo Automático Para Gerar Código Para Novos Pointcuts 29
4.2 Criando uma Extensão para a Declaração @onWeaving 31
4.2.1 Estendendo o Analisador Léxico 31
4.2.2 Estendendo o Analisador Sintático 32
4.2.3 Adicionando um Novo Nó AST 33
4.3 Adaptação do Sistema para Permitir a Definição de Pointcuts em Tempo de Costura 35
4.3.1 Criação de um Novo Visitor 35
4.3.2 Alterando a Sequência de Passos do Compilador 37
4.3.3 Alterações na Fábrica Abstrata 39
4.3.4 Modificação da Classe de Sintaxe 40
4.3.5 Modificação do Visitor 42

5 Testes e Avaliação **43**
5.1 Validação dos Resultados 43
5.2 Avaliação do Trabalho 51

6 Conclusões e Trabalhos Futuros **52**
6.1 Conclusões 52
6.2 Problemas Encontrados e Trabalhos Futuros 53

Referências Bibliográficas **55**

Apêndice A Gramática da Linguagem XAJ **58**

Apêndice B Gramática da Linguagem AspectJ **62**

Apêndice C Gramática da Linguagem Java **75**

Lista de Figuras

2.1 Requisitos transversais espalhados em um módulo do programa. 7
2.2 Exemplo de uso da linguagem AspectJ. 7
2.3 Operações *Polyglot Parser Generator*. 12
2.4 Diferença entre costura em *Jimple* e em *bytecode*. 13
2.5 Classe de sintaxe para definir a extensão *MultiIntro*. 16
2.6 Exemplo de uso da extensão *MultiIntro*. 17

3.1 Diagrama de classes para o exemplo. 22
3.2 Função para alteração de cor com a chamada a *repaint*. 22
3.3 Solução elegante para o problema apresentado. 23
3.4 Código decompilado por DAVA. 24
3.5 Classe de sintaxe para o *contains*. 25
3.6 Gramática para o *contains*. 26
3.7 Bloco *OnWeaving* para o *contains*. 27

4.1 Novo esquema para uma *syntaxclass*. 30
4.2 Extensão do *lexer*. 32
4.3 Extensão para suportar a definição de uma *syntaxclass*. 32
4.4 Extensão para suportar a construção @onWeaving. 34
4.5 Interface para a classe AST da construção @onWeaving. 35
4.6 Interface para as classes visitáveis. 35
4.7 Classe para representar a AST da construção @onWeaving. 36
4.8 Visitor para a construção @onWeaving. 38
4.9 Passos para chamar o Visitor. 39
4.10 Alteração e adição das declarações na interface *SCNodeFactory*. 40
4.11 Alteração e adição das implementações na classe *SCNodeFactory_c*. 40
4.12 Principais modificações para SyntaxClassBody_c.java. 41
4.13 Adicionando método na interface Visitor. 42

4.14 Implementando o método da interface Visitor. 42

5.1 Classe Figure. 44
5.2 Classe Line. 45
5.3 Classe Rectangle. 45
5.4 Classe Editor. 46
5.5 Classe Window. 47
5.6 Aspecto InsertRepaint. 47
5.7 Classe Editor decompilada por DAVA (Parte 1). 48
5.8 Classe Editor decompilada por DAVA (Parte 2). 49
5.9 Método *moveFigureAll*. 50

Lista de Tabelas

2.1 Designadores de *pointcut* nativos de AspectJ (Parte 1) 9
2.2 Designadores de *pointcut* nativos de AspectJ (Parte 2) 10
2.3 Designadores de *pointcut* de *eaj* 14

5.1 Diferenças entre as abordagens para *contains* 51

Resumo

TREVENZOLI, Cleverson Carneiro, M.Sc., Universidade Federal de Viçosa, abril de 2013. **Um mecanismo para extensibilidade de AspectJ em tempo de costura**. Orientador: Vladimir Oliveira Di Iorio. Coorientadores: Alcione de Paiva Oliveira e José Luis Braga.

Linguagens de domínio específico orientadas a aspectos (DSALs) são linguagens de programação com funcionalidades orientadas a aspectos especialmente projetadas para resolver problemas de domínios específicos. O uso destas pode trazer diversas vantagens como a melhora da produtividade e diminuição dos custos na manutenção. Linguagens extensíveis são uma maneira de embutir DSALs em linguagens orientadas a aspectos de propósito geral. A linguagem XAJ (*eXtensible AspectJ*) é uma linguagem extensível baseada em AspectJ, que usa o conceito de classes de sintaxe, unidades que estendem classes com definições de sintaxe, construindo especificações modulares para extensões. XAJ é construída com base no *abc* (*AspectBench Compiler*), um compilador que possui suporte adequado para permitir extensões de linguagem. Visando aumentar o poder de expressividade de AspectJ, diminuindo as limitações e restrições da linguagem, como o conjunto de designadores de *pointcut* fixo e não-extensível, apresentamos um mecanismo de geração automática de código, processável em tempo de costura, para a classe de sintaxe de XAJ. O mecanismo possibilita a definição de novos designadores de *pointcut* pelo usuário mas ainda se limita a extensões que usam apenas informações estáticas do programa base para a geração de código. Essa dissertação descreve problemas encontrados e solucionados na implementação em XAJ desse mecanismo geral e demostra um ganho pela automação de código no processo de extensibilidade, retirando do usuário a necessidade de modificar várias classes do compilador à mão.

Abstract

TREVENZOLI, Cleverson Carneiro, M.Sc., Universidade Federal de Viçosa, april of 2013. **A mechanism for the extensibility of AspectJ at weave time.** Adviser: Vladimir Oliveira Di Iorio. Co-advisers: Alcione de Paiva Oliveira and José Luis Braga.

Domain specific aspect languages (DSALs) are programming languages with aspect-oriented features specially designed to solve specific domains. The use of DSALs may offer several advantages such as improved productivity and reduced costs in maintenance. Extensible languages are a way to embed DSALs on general purpose aspects oriented languages. The XAJ language (eXtensible AspectJ) is an extension of AspectJ which uses the concept of syntax classes, units that extend classes with syntax definitions, building modular specifications for extensions. XAJ is built based on AspectBench, a compiler that has adequate support to allow language extensions. Aiming to increase the power of expressiveness of AspectJ, reducing the limitations and restrictions of the language as a set of pointcuts designators fixed and non-extensible, we present a mechanism for automatic code generation at weave time for XAJ syntax classes. The mechanism allows the definition of new pointcut designators by the users but it is still restrict to extensions that use only static information from the base program. This thesis describes problems that were found and solved, when implementing in XAJ this general mechanism and demonstrates a gain for the automation of code in the process of extensibility, removing the need for the user to modify several classes of a specific compiler by hand.

Capítulo 1

Introdução

O paradigma orientado a aspectos surgiu com o intuito de resolver problemas específicos de modularidade, encapsulando os interesses transversais de uma aplicação, o que evita o entrelaçamento e o espalhamento de código. Para tal, adiciona recursos de orientação a aspectos a diferentes linguagens de programação de propósito geral. Esta separação de interesses ajuda os desenvolvedores a lidar com complexidade de software e reutilização. Entretanto, algumas das características do desenvolvimento de software orientado a aspectos impõem restrições à evolução do software, levando ao problema conhecido como Paradoxo da Evolução do Desenvolvimento de Software Orientado a Aspectos (*AOSD Evolution Paradox*) [Tourwe et al., 2003].

Particularmente, AspectJ, uma linguagem orientada a aspectos de relevância industrial e acadêmica, ainda possui muitas restrições no mecanismo de definição de *pointcuts*, sendo seu conjunto de designadores limitado e não-extensível, o que torna difícil construir soluções elegantes para alguns problemas e diminui significativamente a expressividade da linguagem. Algumas extensões para propósitos específicos foram desenvolvidas, dentre elas, *eaj* [Avgustinov et al., 2005], que estende AspectJ adicionando novos designadores de *pointcut* e, consequentemente, novos recursos. O problema de extensões como essa é justamente o propósito específico, que pode não atender as necessidades do programador.

Com isso, têm surgido outras extensões e muitos trabalhos com o intuito de criar mecanismos para permitir ao usuário modificar a linguagem, a fim de não deixar que AspectJ se torne obsoleta e demasiadamente restrita, mas poucos com um poder de expressividade grande o bastante para permitir a definição de novos designadores de *pointcut* pelo usuário na linguagem AspectJ. Nesse contexto, foi apresentada a primeira proposta para a linguagem XAJ [Di Iorio et al., 2009], que também estende AspectJ com algumas extensões de propósito específico e oferece recursos para extensão léxica e sintática.

Um exemplo de mecanismo para estender o conjunto de *pointcuts* da linguagem também foi apresentado. Já em Di Iorio et al. [2010], foi desenvolvida a implementação de um novo *pointcut* específico, descrevendo o código que seria processado em tempo de costura e o código residual, processado em tempo de execução. A implementação deste foi conduzida com o objetivo de avaliar as dificuldades para a construção de um mecanismo automático, o qual será abordado neste trabalho.

Este trabalho é uma continuação dos estudos elaborados em XAJ e apresenta os primeiros passos sobre a implementação do mecanismo de geração automática de código para designadores de *pointcut* estáticos, que não necessitam da geração de código residual a ser avaliado em tempo de execução. Com este mecanismo, permitimos ao usuário gerar extensões completas para designadores não-dinâmicos em XAJ, através da escrita de código em uma única unidade modular, automatizando boa parte do processo de extensibilidade, sem a necessidade de modificar vários arquivos do compilador.

1.1 O Problema e a sua Importância

Em 1997, Gregor Kiczales apresentou a idéia de um novo paradigma, com orientação a aspectos, em um artigo inicial da Programação Orientada a Aspectos (POA) [Kiczales et al., 1997]. Em 2000 foi lançada a primeira versão de AspectJ [Kiczales et al., 2001], a primeira linguagem orientada a aspectos. Em sua fase inicial, houve uma empolgação da comunidade científica com as possibilidades da POA. Com o passar do tempo, críticas fortes ao modelo foram surgindo.

Alguns dos problemas inerentes às linguagens orientadas a aspectos são:

1. **Dificuldade de Utilização**: a programação orientada a aspectos não é uma técnica trivial, que segue o mesmo padrão das linguagens convencionais, sendo de difícil compreensão e utilização;

2. **Captura Indevida**: os *wildcards* podem realizar a captura não intencional de *joinpoints*. Um *pointcut* definido com um *wildcard* para todos os métodos que tenham certo padrão de nomenclatura age sobre uma certa quantidade de pontos de junção. Ao criar um método, seu nome pode ser compatível com esse *wildcard*, sem que se tenha essa intenção. O mesmo pode ocorrer ao renomear um método;

3. **Quebra de Encapsulamento**: alguns dos recursos das linguagens orientadas a aspectos podem quebrar o encapsulamento, alterando a hierarquia de classes e definições em geral, além de permitir o acesso a membros privados;

4. **Designadores Limitados**: o conjunto de designadores de *pointcut* das linguagens é fixo e limitado. O mecanismo de definição de *pointcuts* fica restrito à captura de certos *joinpoints*, e outros não têm correspondência;

5. **Fragilidade de *Pointcuts* [Störzer & Koppen, 2004]**: este problema está relacionado com a forte dependência do mecanismo de definição de *pointcuts* com a estrutura do código base e seu conhecimento prévio, o que resulta em uma definição de *pointcut* que possui grande acoplamento com o código objeto;

6. **Depuração**: enquanto no nível sintático o código orientado a aspectos aparece em separado, em nível de execução, ele se encontra junto do restante do código. A inserção de adendos pode se tornar imprevisível, tornando a depuração um dos maiores problemas.

Os itens acima descritos foram adaptados de POA [2012].

O problema dos designadores limitados é o que será abordado no presente trabalho. Este problema desestimula a evolução da linguagem e seu amplo uso. Podemos capturar *joinpoints* de maneira limitada; outros, que podem ser pontos de junção pretendidos pelo programador, não podem ser capturados em linguagens como AspectJ, pois não é possível especificar um algoritmo que os descreva exatamente.

1.2 Motivação

A hipótese levada em consideração é de que a inclusão de mecanismos de definição de novos designadores de *pointcut* pelo usuário possibilitará a total ou parcial solução do problema relacionado ao conjunto limitado dos designadores e, consequentemente, do Paradoxo da Evolução do Desenvolvimento de Software Orientado a Aspectos. Isto ampliará o uso do paradigma e difundirá o mesmo de forma abrangente.

Se for pretensão do programador capturar pontos de junção em que ocorrem a definição de *loops* por comandos como o *while*, o que não é permitido em AspectJ, por exemplo, o próprio programador poderá criar uma extensão na linguagem XAJ [Di Iorio et al., 2009] permitindo que essa captura seja realizada. Assim, o trabalho poderá ser facilitado, os recursos da linguagem aumentarão e o desenvolvimento de software orientado a aspectos poderá evoluir.

Visto que XAJ tem como um dos objetivos a modularidade das extensões, a versão final da implementação da classe de sintaxe deverá conter todas as funcionalidades para permitir a definição de qualquer extensão para a linguagem. O programador poderá definir extensões seguindo regras específicas do compilador de XAJ, o que reduzirá os gastos

com escrita de código e eliminará o processo de modificação do compilador base, o *abc*. A linguagem XAJ já possui alguns mecanismos de automação do processo de extensão embutidos na linguagem. Permanece não implementado na linguagem o mecanismo que gera código em tempo de costura, definindo extensões de natureza estática ou dinâmica.

1.3 Objetivo

Este trabalho tem como objetivo criar um mecanismo de extensão para XAJ, permitindo a geração de código em tempo de costura para designadores de *pointcut* definidos pelo usuário, que usam informações estáticas do programa base. Assim, é possível contornar o problema da geração de código residual para as extensões, inserindo informações em tempo de costura de código, sem deixar todas as verificações para o tempo de execução, diminuindo a sobrecarga.

1.4 Como o Texto está Organizado

O Capítulo 2 define o embasamento teórico para nosso trabalho. Apresenta o paradigma orientado a aspectos e um exemplo de uso, a linguagem AspectJ e seus designadores nativos, o compilador *AspectBench* e as ferramentas em que foi construído, uma extensão para AspectJ e seus benefícios, conceitos de resíduos e código residual e extensibilidade das linguagens orientadas a aspectos, além dos trabalhos relacionados e da linguagem XAJ. A extensibilidade de *pointcuts* em XAJ é apresentada no Capítulo 3, mostrando um exemplo para motivação e sua solução na linguagem. O Capítulo 4 mostra a implementação do mecanismo proposto no Capítulo 3. A validação dos resultados obtidos é feita no Capítulo 5. As conclusões e trabalhos futuros são discutidos no Capítulo 6.

Capítulo 2

Referencial Teórico

Neste capítulo, primeiramente, é apresentada a Programação Orientada a Aspectos e a sua funcionalidade por meio de um exemplo de uso. Na Seção 2.1.1 apresentamos uma das linguagens mais difundidas no paradigma orientado a aspectos, AspectJ, e as limitações impostas pelo seu mecanismo de definição de *pointcuts*. Discutimos também um pouco sobre seus compiladores, mais detalhadamente a respeito do *abc* e as ferramentas em que foi baseado. A seguir, definimos uma extensão de AspectJ e apresentamos alguns dos novos designadores que ela implementa e suas respectivas funcionalidades. Após a abordagem destes conceitos, apresentamos um estudo sobre resíduos, apontando alguns exemplos de designadores que geram resíduos dinâmicos no código final e outros que não os geram, definimos nossa ferramenta de trabalho, a linguagem XAJ, e, por fim, relatamos trabalhos mais diretamente relacionados ao assunto desta dissertação.

2.1 Programação Orientada a Aspectos

A Programação Orientada a Aspectos (POA) surgiu com o objetivo de proporcionar uma modularização adequada de interesses transversais, evitando entrelaçamento e espalhamento de código (*tangling* e *scattering* [Kiczales et al., 1997]). Esses problemas surgem com freqüência quando se implementa interesses transversais usando linguagens orientadas a objetos, tornando o código difícil de escrever, entender, reutilizar e manter. Isso dificulta algumas propriedades desejáveis a toda linguagem de programação [Varejão, 2004]:

1. Redigibilidade: capacidade de redação de programas de forma simples, sem se preocupar com aspectos não relevantes para a solução do problema;

2. Legibilidade: facilidade para se ler e entender um programa;

3. Reusabilidade: possibilidade de reutilizar o mesmo código em outras aplicações, aumentando a produtividade;

4. Modificabilidade: capacidade de alterar o programa em função de novos requisitos, sem que tais modificações impliquem em mudanças em outras partes do programa.

Programas orientados a aspectos resolvem estes problemas, através da separação de interesses. Assim como a programação orientada a objetos é uma forma de modularizar interesses comuns, a programação orientada a aspectos visa a separação dos interesses transversais, focando cada um deles separadamente em uma unidade modular chamada aspecto.

A POA introduz um novo conceito chamado *joinpoint* que é um ponto bem definido no fluxo de controle do programa. Um conjunto de *joinpoints* determina um *pointcut*. Para capturar tais pontos, usamos designadores de *pointcut* que identificam relações entre o *joinpoint* e o programa, como chamada ou execução de um método, acesso ou escrita de um campo, dentre outros.

O entrelaçamento de código ocorre quando em um mesmo módulo temos códigos referentes a propriedades diferentes do sistema. Por exemplo, código de negócio com o de apresentação. O espalhamento de código ocorre quando temos trechos de códigos que desempenham a mesma funcionalidade espalhados pelo sistema, como o referente a acesso a dados presente em vários módulos do mesmo. Parte destas limitações pode ser compensada com o uso de padrões de projetos [Gamma et al., 1995].

São exemplos importantes de requisitos transversais:

- registro de operações (*logging*);
- segurança;
- concorrência;
- persistência de dados;
- controle de transações;
- *caching*;
- *pooling*.

A Figura 2.1 mostra alguns requisitos transversais espalhados em uma unidade modular (a classe *UmaClasse*) do programa. Operações de autenticação e de *logging* são requeridas pelo método *UmMetodo*.

```
public class UmaClasse {
    ... declaração de membros
    public void UmMetodo(...) {
        [autenticação de usuário]
        [registro inicial da operação (logging)]
        ... execução da operação
        [registro final da operação (logging)]
    }
}
```

Figura 2.1. Requisitos transversais espalhados em um módulo do programa.

```
1  public aspect LoggingAspect {
2    pointcut publicMethods() : execution(public * * (..));
3    pointcut mainMethods() : execution(public * main(String[]));
4    pointcut loggableCalls() : publicMethods() && !mainMethods();
5
6    before() : loggableCalls() {
7      System.out.println("Início:␣" + thisJoinPoint.getSignature());
8    }
9
10   after() : loggableCalls() {
11     System.out.println("Final:␣" + thisJoinPoint.getSignature());
12   }
```

Figura 2.2. Exemplo de uso da linguagem AspectJ.

Um exemplo de aspecto implementado em AspectJ para operações de *logging* é mostrado na Figura 2.2. Em suma, o *log* de operações demonstrado deverá:

- No início da execução de todo método público da classe, inserir chamada de início de operação, identificando o método e outros valores não demonstrados no exemplo;
- No final da execução de todo método público da classe, inserir chamada de fim de operação, identificando o método e outros valores não demonstrados no exemplo.

AspectJ é uma linguagem muito importante no contexto de orientação a aspectos, e os compiladores mais importantes para tal linguagem são o *ajc* [AspectJ, 2012] e o *abc* [Avgustinov et al., 2005]. O *abc* (*AspectBench Compiler*) é *open source* e foi criado com o propósito de ser usado para pesquisas na área além de suportar a fácil extensibilidade. Sendo o conjunto de designadores de pontos de junção em AspectJ fixo e limitado, temos que a linguagem em si também é limitada. Assim, um compilador extensível se torna de grande importância para a linguagem.

2.1.1 AspectJ e suas Limitações

AspectJ é uma linguagem idealizada por Gregor Kiczales, que implementa o paradigma orientado a aspectos. Foi desenvolvida por uma divisão de pesquisa da *Xerox*, denominada *Xerox PARC*. Hoje é mantida pela *Eclipse Foundation*.

Muitos recursos para AspectJ foram e continuam sendo desenvolvidos, mas a linguagem ainda possui muitas limitações. Seus designadores de *pointcut* incluem chamada ou execução de métodos ou construtores, inicialização de uma classe ou objeto, leitura de campos e acesso à escrita, tratamento de exceções, dentre outros apresentados nas Tabelas 2.1 e 2.2 [Eclipse, 2012]. Eles não incluem *loops* (*while* e *for*), chamadas de classe mãe (*super*), cláusulas *throws* e declarações múltiplas, por exemplo.

Por mais que existam diversas extensões, sempre haverá alguma outra para complementar uma linguagem. Com AspectJ isso não é diferente. Podemos pensar em códigos complexos e pontos no programa ainda inatingíveis e que poderiam ser *joinpoint shadows* (pontos de junção potenciais) para algumas finalidades, condições a serem testadas, cuja correspondência só seria aceita em algum *pointcut* se o valor booleano for verdadeiro, estruturas de repetição a se querer capturar no código, dentre muitas outras possibilidades de problemas para que novas extensões os resolvam.

2.2 O Compilador AspectBench

O *abc* (*AspectBench Compiler*) é um compilador *open source* para AspectJ, projetado com o objetivo de ser usado em pesquisas na área. Criado como compilador de fácil extensibilidade, ele se torna bastante útil nos projetos de desenvolvimento de software, permitindo modificações a um nível mais baixo.

O *abc* é uma implementação completa do AspectJ com algumas diferenças para o compilador original *ajc*, que visam torná-lo fácil de modificar, criando e adaptando extensões, e permitir otimizações da linguagem. Seu núcleo oferece uma interface construída sobre o Polyglot (Seção 2.2.1). Sua infra-estrutura é construída sobre o Soot (Seção 2.2.2), para a otimização de Java.

Elaborar extensões para o *abc* nos obriga a estudar e aprender seu mecanismo funcional, a fim de obter um conhecimento sobre o *frontend*, o *backend* e a maneira que ambos se relacionam, tornando o *abc* um compilador extensível. Na primeira parte desta seção, discutimos um pouco sobre o Polyglot, seu funcionamento e as facilidades que ele permite ao *abc* e, na segunda, analisamos o Soot. O Polyglot gera uma AST anotada com informação de tipo e contém um mecanismo de extensão gramatical que proporciona a separação da gramática original AspectJ da adicionada por uma extensão. O Soot,

Tabela 2.1. Designadores de *pointcut* nativos de AspectJ (Parte 1)

Métodos e Construtores	
call(Signature)	toda chamada a algum método ou construtor correspondente a *Signature* no local da chamada
execution(Signature)	toda execução de algum método ou construtor correspondente a *Signature*
Campos	
get(Signature)	toda referência a algum campo correspondendo a *Signature*
set(Signature)	toda atribuição a algum campo correspondente a *Signature*. O valor atribuído pode ser exposto com um pointcut *args*
Manipuladores de Exceção	
handler(TypePattern)	todo tratamento de exceção para algum *Throwable* em *TypePattern*. O valor da exceção pode ser exposto com um pointcut *args*
Adendo	
adviceexecution()	toda execução de algum fragmento de *advice*
Inicialização	
staticinitialization(TypePattern)	toda execução de um inicializador estático para algum tipo em *TypePattern*
initialization(Signature)	toda inicialização de um objeto quando o primeiro construtor chamado no tipo correspondente a *Signature*, abrangendo o retorno da chamada do construtor *super* para o retorno do primeiro construtor chamado
preinitialization(Signature)	toda pré-inicialização de um objeto quando o primeiro construtor chamado no tipo correspondente a *Signature*, abrangendo a entrada do primeiro construtor chamado para a chamada ao construtor *super*

utilizando-se de uma representação intermediária, facilita a costura de extensões além de analisar e transformar o código final para *bytecode*, inserindo algumas otimizações.

2.2.1 Polyglot

Polyglot [Nystrom et al., 2003] é um *framework* utilizado no *frontend* do *abc* de modo a facilitar e permitir extensibilidade, através da modificação da gramática de acordo com o que se pretende, proporcionando todas as verificações semânticas exigidas pela linguagem e atendendo os critérios de simplicidade, modularidade e proporcionalidade em uma

Tabela 2.2. Designadores de *pointcut* nativos de AspectJ (Parte 2)

Estrutura Léxica (Lexical)	
within(TypePattern)	todo join point do código definido por um tipo em *TypePattern*
withincode(Signature)	todo join point do código definido por um método ou construtor correspondente a *Signature*
Verificações de Instanciação e Exposição de Contexto	
this(Type or Id)	todo join point quando o objeto atualmente em execução é uma instância de *Type* ou tipo de *Id*
target(Type or Id)	todo join point quando o objeto alvo em execução é uma instância de *Type* ou tipo de *Id*
args(Type or Id, ...)	todo join point quando os argumentos são instâncias de *Type* ou tipo de *Id*
Fluxo de Controle	
cflow(Pointcut)	todo join point no fluxo de controle de cada join point *P* capturados por *Pointcut*, incluindo o próprio P
cflowbelow(Pointcut)	todo join point sob o fluxo de controle de cada join point *P* capturados por *Pointcut*; não incluindo o próprio *P*
Condicional	
if(Expression)	todo join point quando o booleano *Expression* é *true*
Combinação	
! Pointcut	todo join point não capturado por *Pointcut*
Pointcut0 && Pointcut1	todo join point capturado por ambos *Pointcut0* e *Pointcut1*
Pointcut0 || Pointcut1	todo join point capturado por *Pointcut0* e/ou *Pointcut1*
(Pointcut)	todo join point capturado por *Pointcut*

grande variedade de extensões para a sintaxe e sistema de tipos. Ele é estruturado como uma lista de passos que reescrevem a AST e constrói estruturas auxiliares, tais como uma tabela de símbolos e o sistema de tipos. O *abc* não faz qualquer alteração neste componente, somente se baseia nele.

Polyglot age segundo a sequência descrita abaixo:

1. analisa o código-fonte Java da árvore de sintaxe abstrata (AST);

2. executa todas as verificações estáticas exigidas pela linguagem Java em um número de passos e reescreve a árvore, de forma que a saída do Polyglot é uma AST Java

anotada com informação de tipo;

3. escreve de volta para um arquivo fonte Java tal AST anotada.

De modo a reconhecer AspectJ, que é uma extensão de Java, foi desenvolvida uma extensão da sintaxe de Java no Polyglot. Vários de seus recursos o tornam adequado para criar extensões, e também ajudam a fazer essas extensões em si extensíveis. O Polyglot inclui um gerador de analisador sintático, o *Polyglot Parser Generator* - PPG [Brukman & Myers, 2008] para gramáticas extensíveis. O PPG permite que uma nova gramática seja especificada através de modificações de uma gramática já existente, onde estas modificações são dadas em um arquivo de especificação em separado (e não no arquivo da gramática original), usando palavras-chaves para importar a gramática original que se pretende estender (palavra-chave *import*) e estendendo o não-terminal apropriado (através do uso de *extend*).

Independentemente de estender as alternativas para não-terminais existentes, o *Polyglot Parser Generator* também permite que um usuário elimine produções, transfira produções de um não-terminal a outro, e substitua a produção de um determinado não-terminal.

Algumas das operações disponibilizadas pelo PPG são demonstradas na Figura 2.3. O arquivo *X* especifica uma gramática cujo não-terminal *S* pode gerar os terminais *a*, *b* ou *c*. O arquivo *Y* especifica outra gramática diferente da primeira mas que importa as produções gramaticais de *X* (*include*) e estende *S* (*extend*) possibilitando a geração dos terminais *d* e *e* a partir de *S*. A gramática especificada no arquivo *Z* importa as produções gramaticais de *Y* e elimina (*drop*) as produções *b* e *d* a partir de *S*. O resultado (*Result*) será uma gramática definida pela produção a partir de *S* dos símbolos terminais *a*, *c* e *e*.

2.2.2 Soot

Soot Vallée-Rai et al. [1999] é um *framework* para análise e transformação Java *bytecode*, usado como o *backend* do *abc*. A mais importante vantagem do uso de Soot, tanto para o desenvolvimento do próprio *abc* quanto para estender a linguagem, está na sua representação intermediária *Jimple*. *Jimple* é uma representação de *bytecode* com informação de tipos, baseada em uma arquitetura de três endereços, que dispensa preocupação a respeito das operações implícitas sobre a pilha de computação (todas as operações são expressas em termos de variáveis explícitas). Por ser tipada, é possível saber o tipo de cada expressão ou variável. Por utilizar código baseado na arquitetura de três endereços (*3-address code*), toda instrução é o mais simples possível, a maioria sendo da forma

File **X**	File **Y**
$S ::= a$ $\mid b$ $\mid c$	*include* **X** *extend* $S ::= d$ $\mid e$
File **Z**	**Result**
include **Y** *drop* $S ::= b$ $\mid d$	$S ::= a$ $\mid c$ $\mid e$

Figura 2.3. Operações *Polyglot Parser Generator*.

$$x = y \ op \ z.$$

Jimple também é compacta, ou seja, o número de instruções de *Jimple* é consideravelmente menor do que o de *bytecode*, porém mantendo o mesmo poder representativo.

O Soot fornece módulos para conversão em *Jimple*, Java *bytecode* e código-fonte Java. Além disso, ele inclui um decompilador, DAVA [Miecznikowski & Hendren, 2002], que é muito útil para visualizar o efeito dos aspectos e das extensões AspectJ no código gerado.

Em vez de representar computações com uma pilha implícita, cada instrução *Jimple* manipula explicitamente variáveis locais especificadas, o que simplifica a costura de adendos. Se ele fosse costurado em *bytecode* diretamente (como o *ajc* faz), o *weaver* necessitaria de considerar o efeito do código costurado sobre a pilha de execução implícita e gerar código adicional para organizar o conteúdo da pilha. Além disso, quando os valores do *joinpoint shadow* são necessários como parâmetros para os adendos, eles estão prontamente disponíveis nas variáveis locais; o *weaver* não tem que examinar minuciosamente através da pilha de computação para encontrá-los, o que implicaria em perda de rendimento.

Como entrada, Soot pode manipular arquivos *.class* e arquivos fonte Java. Para converter *bytecode* para *Jimple*, Soot introduz uma variável local para representar explicitamente cada localização de pilha, divide as variáveis para separar utilizações independentes da mesma localização, e infere um tipo para cada variável. Para converter o

código fonte para *Jimple*, Soot primeiro usa Polyglot para construir uma AST com informações sobre tipo e, em seguida, gera o código *Jimple* a partir desta. Normalmente, após todo o processamento, Soot converte o código *Jimple* em *bytecode* e o grava em arquivos *.class*. Este processo também inclui otimizações importantes para a geração de *bytecode* eficiente.

A Figura 2.4 compara a costura feita pelo *abc* com a feita pelo *ajc*. Note que o *bytecode* costurado usando a representação *Jimple* é notavelmente mais simplificado que a costura direta feita pelo *ajc*.

```
public int f(int x,int y,int z)
{
    return bar(x, y, z);
}
```

(a) base Java code

```
public int f(int x,int y,int z)
0:  aload_0
1:  iload_1
2:  iload_2
3:  iload_3
4:  istore %4
6:  istore %5
8:  istore %6
10: astore %7
12: invokestatic
     A.aspectOf ()LA;
15: aload  %7
17: invokevirtual
     A.ajc$before$A$124 (LFoo;)V
20: aload %7
22: iload %6
24: iload %5
26: iload %4
28: invokevirtual Foo.bar (III)I
31: ireturn
```

(b) direct weaving into bytecode (*ajc*)

```
public int f(int,int,int)
{ Foo this;
   int x, y, z, $i0;
   A theAspect;

   this := @this;
   x := @parameter0;
   y := @parameter1;
   z := @parameter2;
   theAspect = A.aspectOf();
   theAspect.before$0(this);
   $i0 = this.bar(x, y, z);
   return $i0;
}
```

(c) weaving into Jimple (*abc*)

```
public int f(int x,int y,int z)
0:  invokestatic A.aspectOf ()LA;
3:  aload_0
4:  invokevirtual
     A.before$0 (LFoo;)V
7:  aload_0
8:  iload_1
9:  iload_2
10: iload_3
11: invokevirtual Foo.bar (III)I
14: ireturn
```

(d) bytecode generated from Jimple (*abc*)

Figura 2.4. Diferença entre costura em *Jimple* e em *bytecode*.

Quando utiliza-se o Polyglot como um *frontend* para o Soot, o módulo *Java2Jimple* (*Java To Jimple*) dentro deste compila a AST final em representação intermediária *Jimple* ao invés de ser escrita para um arquivo fonte Java. Portanto, no *abc*, o passo final do Polyglot separa o programa AspectJ em Java puro (que é passado para *Jimple* no Soot) e instruções para o *backend*.

2.3 A extensão *eaj*

Extensible AspectJ (*eaj*) é uma extensão de AspectJ desenvolvida pelos próprios criadores do *abc*. Teve como principal intuito demonstrar as funcionalidades desse compilador e quais os procedimentos para se criar uma extensão a partir dos seus recursos. A extensão *eaj* adicionou designadores de *pointcut* à linguagem, dentre eles os definidos na Tabela 2.3. A extensão permite, entre outras coisas, a captura de *pointcuts* em cláusulas *throw*, o que não é possível em AspectJ.

Outros designadores de *pointcut* que *eaj* adicionou à linguagem AspectJ são: *let*, *lock*, *unlock*, *cflowdepth*, *cflowbelowdepth* e *maybeShared*. Eles não serão apresentados neste trabalho, sendo citados apenas para efeito didático.

Tabela 2.3. Designadores de *pointcut* de *eaj*

Extensão eaj	
cast(TypePattern)	toda conversão para algum tipo em *TypePattern*
contains(Pointcut)	todo join point em cujo contexto ocorrem outros definidos em *Pointcut*
throw(TypePattern)	todo join point em que ocorre um lançamento de exceção para algum tipo em *TypePattern*
Global:<ClassPattern>:<Pointcut>	substitui o pointcut de cada declaração de adendo em cada aspecto cujo nome corresponde a *ClassPattern* com a conjunção do pointcut original e do global definido por *Pointcut*
arrayget()	toda referência a um campo do tipo arranjo
arrayset()	toda atribuição a um campo do tipo arranjo

2.4 Resíduos e Código Residual

Alguns designadores de *pointcut* necessitam da geração de código para a obtenção de informações em tempo de execução. Esse código é gerado durante a fase de *weaving*. O *abc* é otimizado para retirar códigos desnecessários que podem ser avaliados estaticamente. O designador *if*, por exemplo, necessita de informações sobre o valor lógico de uma expressão (se é um booleano *true* ou *false* em sua avaliação). Tal avaliação nem sempre pode ser determinada com as informações estáticas. Logo, é gerado um código para fazer uma verificação em tempo de execução. Este é chamado código residual e o

resíduo gerado é dito dinâmico. E designadores que podem gerar resíduos em tempo de costura, tais como *if*, *cflow*, *cflowbelow*, *target* e *this*, também são ditos dinâmicos.

Outros designadores não necessitam de tal geração de código, sendo que todas as avaliações a respeito do mecanismo de funcionamento destes sempre podem ser obtidas através de informações estáticas ou, até mesmo, nem necessitem de informação alguma, somente designando e correspondendo a pontos de junção potenciais no código. Designadores como *call*, *execution*, *within*, *get* e *set*, são exemplo de designadores não-dinâmicos.

2.5 A Linguagem XAJ

Inicialmente proposta em Di Iorio et al. [2009], a linguagem XAJ teve sua primeira versão e um compilador (*xajc*) implementados em 2009 [Reis et al., 2009]. A linguagem XAJ (*eXtensible AspectJ*) é uma extensão da linguagem AspectJ, baseada no compilador *AspectBench*, que permite estender sua sintaxe concreta. Ela foi inspirada nas linguagens XMF [Clark., 2009] e XJ [Clark et al., 2008], apresentando o mesmo conceito de classes de sintaxe de XMF e XJ. Porém, as extensões definidas na linguagem XAJ são usadas como se fossem elementos da linguagem base. A finalidade de criar extensões e embarcá-las em uma linguagem de próposito geral é oferecer recursos mais apropriados à linguagem para expressar um problema em questão.

Classes de sintaxe são unidades sintáticas compiláveis para a definição de extensões para a linguagem XAJ. Elas aumentam a modularidade, sendo que toda a definição relativa à construção (sintaxe e semântica) está encapsulada em uma mesma unidade de compilação. XAJ também aumenta a portabilidade das extensões, pois elas são definidas na própria linguagem, não dependendo de ferramentas específicas em sua implementação.

Classes de sintaxe são definidas em arquivos *.java* por meio do uso da palavra-chave *syntaxclass*, o que as diferencia das classes usuais (definidas por *class*). Igualmente as classes de Java, classes de sintaxe também podem usar os modificadores de acesso *public* ou *private*, conter funções e atributos. Além disso, elas podem conter uma declaração sintática e um método especial, denominado *desugar*, para determinar a semântica das novas construções, estabelecendo uma tradução para AspectJ puro, usando programação generativa. Após a AST ser construída, contendo nós de extensão representados pelas classes de sintaxe, o método *desugar* é executado em tempo de compilação e os nós das novas construções são substituídos por uma sub-árvore cujos nós têm somente construções na linguagem AspectJ.

Resumindo, as classes sintáticas servem para:

- Especificar a sintaxe de novas construções, modificando a gramática de AspectJ,

```
1  public syntaxclass MultiIntro {
2    @grammar extends intertype_member_declaration
3      MultiIntro ->
4        modifiers = modifiers_opt
5        returnType = type
6        className = identifier
7        "+." methodName = identifier
8        "(" params = formal_parameter_list_opt ")"
9        introducedCode = block;
10   }
11
12   public Node desugar(Context ct, NodeFactory ft, TypeSystem ts) {
13     ClassDecl cd = ctx.getClass(className);
14     ClassDecl sub[] = cd.getSubClasses();
15     List list = nf.TypedList(ClassMember.class);
16     for(ClassDecl x : sub)
17       list.add(nf.IntertypeMethodDecl(modifiers, returntype,
18                  x, methodName, params, block));
19     return list;
20   }
21 }
```

Figura 2.5. Classe de sintaxe para definir a extensão *MultiIntro*.

acrescentando novas produções ou modificando as existentes;

- Definir a semântica de novas construções ou modificar a semântica de construções existentes;
- Representar as novas construções na árvore de sintaxe abstrata.

É importante observar que, dessa forma, XAJ não permite excluir produções da linguagem original.

A Figura 2.5 apresenta a classe de sintaxe MultiIntro, para criar a extensão *MultiIntro* discutida em Di Iorio et al. [2009]. Ela estende o mecanismo de declarações inter-tipo de AspectJ adicionando a possibilidade de modificar o código a ser inserido, em tempo de compilação, através do método *desugar*. Esta extensão elimina a redundância relacionada a declarações inter-tipo para casos em que a declaração deve ser inserida em várias classes, mas seguindo um padrão comum. Por exemplo, na Figura 2.6, AspectJ permite adicionar o método *accept* do padrão de projeto *Visitor* para uma superclasse *Base* e suas subclasses através do uso da construção *Base+*. Assim, *MultiIntro* permite adicionar o mesmo método dando a possibilidade de modificar seu nome (*acceptClassX*, *acceptClassY*). Esta classe de sintaxe contém apenas o método *desugar* e a declaração sintática que define a sintaxe da extensão criada.

Uma declaração sintática é declarada usando o comando especial @grammar, como apresentado na Figura 2.5, e tem a semântica de modificar a gramática da linguagem XAJ,

```
1 void Base+.accept(Visitor v) {
2   v.visit(this);
3 }
```

Figura 2.6. Exemplo de uso da extensão *MultiIntro*.

adicionando novas produções, novos não-terminais ou modificando o comportamento de produções existentes. Existem três tipos de declaração sintática:

- Declaração sintática simples: utiliza não-terminais existentes (*using*);
- Declaração sintática estendida: estende um não-terminal existente (*extends*);
- Declaração sintática de sobrecarga: substitui um não-terminal existente (*overrides*).

A extensão *MultiIntro* usa declaração sintática estendida, caracterizada pelo uso da palavra-chave *extends* depois de @grammar.

A semântica das extensões definidas pelas declarações sintáticas é dada no método *desugar*. Este método é executado em tempo de compilação e irá substituir o nó que representa a extensão por novas construções da linguagem AspectJ, usando programação generativa. No método *desugar* da Figura 2.5, uma declaração intertipo para cada subclasse de *className* é criada e adicionada em uma lista de nós que é retornada no fim do método.

Em algumas aplicações, pode ser necessário mais de um passo para traduzir uma extensão, ou informações devem ser colhidas antes da tradução ser realizada. Nesses casos específicos, uma versão alternativa do método *desugar* pode ser usada, o qual recebe um parâmetro adicional com o valor do passo atual.

O número de passos que uma classe de sintaxe necessita deve ser especificado na mesma, usando o comando @numberOfPasses.

Uma classe de sintaxe tem uma única declaração sintática e no máximo um uso do comando @numberOfPasses. Se o comando @numberOfPasses não for usado, assume-se que a classe de sintaxe necessita de um único passo e a versão alternativa do método *desugar* não será chamada.

A linguagem XAJ oferece duas formas para especificar o escopo das extensões definidas pelas classes de sintaxe, a importação da sintaxe e a importação de símbolos, que funcionam de forma semelhante ao *import* da linguagem AspectJ e aumentam a modularidade das extensões. No primeiro caso, o escopo é definido pela palavra-chave *importsyntax*, cuja semântica é especificar que a sintaxe da extensão definida pela classe de sintaxe importada pode ser usada no escopo da importação. Já a importação de símbolos é usada

por classes de sintaxe para importar novos não-terminais definidos por outras classes de sintaxe, permitindo que classes de sintaxe usem em sua declaração sintática a definição da sintaxe de extensões definidas em outras classes de sintaxe. A importação de símbolo só define que o nome deste símbolo pode ser usado dentro do escopo que foi importado.

A extensibilidade de *pointcuts* em XAJ é fonte de estudos para vários trabalhos. O trabalho desenvolvido nesta dissertação se refere apenas à extensibilidade de XAJ em tempo de costura.

2.6 Trabalhos Relacionados

Nesta seção serão discutidos trabalhos que envolvem o paradigma orientado a aspectos e a extensibilidade de linguagens, assuntos que estão mais diretamente relacionados ao trabalho apresentado por esta dissertação.

Muita pesquisa tem sido conduzida com o objetivo de superar as atuais limitações das linguagens orientadas a aspectos e criar mecanismos, extensões ou mesmo novas linguagens, para permitir que o usuário modifique características nativas das linguagens e/ou desenvolva novos recursos para as mesmas. Um exemplo é a linguagem Josh [Chiba & Nakagawa, 2004], similar a AspectJ, contendo recursos de extensibilidade. Em Josh, novos designadores de *pointcut* podem ser usados contanto que sigam a sintaxe de *pointcut* padrão. A semântica desses novos designadores é definida por um método estático em Java, executado durante o tempo de costura. Uma biblioteca de reflexão de tempo de compilação chamada Javassist [Chiba & Nishizawa, 2003] fornece métodos para inspecionar a estrutura estática do programa. Se algum código residual é necessário, ele pode ser inserido explicitamente no programa através de um *framework* de manipulação de *bytecode*. Diferente de AspectJ, Josh não fornece um designador como *args* para exposição de contexto, mas fornece variáveis especiais disponíveis no corpo do adendo para esse propósito.

Em Breuel & Reverbel [2007], os autores propõem uma abordagem denominada *Join Point Selectors*, um mecanismo de extensão simples para o enriquecimento de linguagens de *pointcut* com construções que desempenham o papel de "novos *pointcuts* primitivos". Um programador pode decidir qual código é executado durante o tempo de costura e qual código deve ser executado em tempo de execução. A implementação fornecida para seletores de pontos de junção é desenvolvida como uma extensão do *framework* JBOSS AOP [JBoss, 2012] e o código que será executado durante o tempo de execução é definido estaticamente.

SCoPE (*Static Conditional Pointcut Evaluation*) [Aotani & Masuhara, 2007] é de-

senvolvida como uma extensão do *abc* [Avgustinov et al., 2005], um compilador extensível baseado em ferramentas que facilitam a extensão da linguagem AspectJ, bastante usado em pesquisas na área. SCoPE não estende a linguagem base e suporta *pointcuts* definidos pelo usuário, permitindo ao programador escrever um *pointcut* que analisa um programa usando um *pointcut* condicional (*if*) com bibliotecas de reflexão introspectivas. Assim, testes de tempo de execução para esse *pointcut*, que podem ser processados em tempo de costura por meio de informações estáticas, são automaticamente eliminados pelo compilador *AspectBench*.

Com o mesmo objetivo de superar as limitações de linguagens orientadas a aspectos, surgiu XAJ [Di Iorio et al., 2009]. XAJ é uma extensão de AspectJ que oferece recursos para estender a própria linguagem. Foi implementada como uma extensão do compilador *abc* e usa diversos dos seus recursos, tanto no *frontend* quanto no *backend*. Através da definição de classes de sintaxe (*syntaxclass*), é possível modificar a linguagem adicionando novas funcionalidades. As classes de sintaxe oferecem um mecanismo de extensão léxica e gramatical e, por meio de um método especial denominado *onWeaving*, permitem definição de código a ser executado em tempo de costura. Esse método pode ser chamado em novos pontos de junção definidos pelo programador, sendo usadas técnicas de programação generativa para permitir a tradução para a linguagem AspectJ pura. Esse mecanismo visa proporcionar uma separação clara de processamento em tempo de execução e em tempo de costura.

Em termos de questões de implementação, SCoPE tem significativa semelhança a XAJ. Uma diferença importante entre SCoPE e XAJ é que o primeiro tenta automaticamente eliminar, tanto quanto possível testes de tempo de execução em *pointcuts* definidos pelo usuário. Este último, por outro lado, dá ao programador um poder de decidir exatamente que código é executado durante o tempo de costura e de execução.

Os recursos de XAJ foram apresentados em Di Iorio et al. [2009] apenas como uma proposta de funcionamento. Na sequência do trabalho, em Di Iorio et al. [2010], são apresentados detalhes da implementação de um novo designador de *pointcut* específico, denominado *IsType*, com o objetivo de analisar se uma variável possui um determinado tipo, com otimizações e eliminação de código residual, sempre que possível, usando os recursos do compilador *AspectBench*. Essa extensão caracteriza um designador de *pointcut* dinâmico, que pode gerar código residual e, portanto, necessita de recursos mais sofisticados que os empregados por SCoPE, por exemplo. A compreensão das tarefas necessárias para a implementação deste designador específico aumentou o conhecimento da equipe sobre as ferramentas utilizadas, sendo, portanto, fundamental para o desenvolvimento de um mecanismo genérico em XAJ para a extensão do conjunto de designadores de *pointcut*.

Outro trabalho relacionado a XAJ, descrito em Di Iorio et al. [2011], discute uma metodologia proposta para especificar claramente a separação do código que será executado em tempo de costura do código que será processado em tempo de execução. É apresentado um paralelo entre o mecanismo de macros, para extensão de sintaxe em linguagens convencionais, com o mecanismo de extensão de pointcuts de XAJ, criando o termo *macro em tempo de costura*.

Neste trabalho, apresentamos a primeira implementação no compilador de XAJ de um mecanismo automático para gerar código em tempo de costura para novos designadores de *pointcut* definidos pelo usuário, restringindo-se àqueles que necessitam apenas de informações estáticas do programa base. São propostas alterações para algumas construções da linguagem XAJ definidas em Di Iorio et al. [2009] e Di Iorio et al. [2011], de modo a facilitar o trabalho de implementação.

Capítulo 3

Extensibilidade de Pointcuts em XAJ

Neste capítulo, definiremos a motivação pela qual este trabalho foi conduzido, descrevendo um problema encontrado em um exemplo de aplicação que possui um determinado interesse transversal que não é modularizável pelos recursos atuais da linguagem AspectJ. Mostraremos também uma solução elegante para a modularização deste interesse utilizando *eaj* e XAJ.

3.1 Exemplo para Motivação

Suponha um editor de imagens baseadas em figuras geométricas, cujo diagrama de classes está parcialmente representado na Figura 3.1. O programa contém métodos para desenhar quadrados, linhas, círculos, dentre outros elementos e um método para redesenhar a tela da aplicação chamado *repaint*, que chama, sucessivamente, o método *redraw* sobre cada elemento figura desenhado, até atualizar a janela da aplicação por completo. A chamada para *repaint* deve ser realizada sempre que um atributo de algum elemento *Figure* for alterado, para mostrar as modificações na tela para o usuário, seja alteração de posição, cor ou outros atributos.

Desejamos identificar se há, dentro de quaisquer métodos, modificações sobre atributos de *Figure*. Caso positivo, devemos garantir que seja inserida apenas uma única chamada a *repaint* ao final da execução destes métodos. Assim, a tela seria atualizada, apresentando as alterações feitas. Uma solução elegante consiste em modularizar este interesse transversal (de atualização de tela) em um aspecto, não deixando a chamada desse código espalhada em todo lugar onde são feitas modificações de atributos de *Figure*. A linguagem AspectJ não possibilita especificar um *pointcut* que realize tal tarefa ou mesmo escrever um algoritmo para selecionar os *joinpoints* pretendidos.

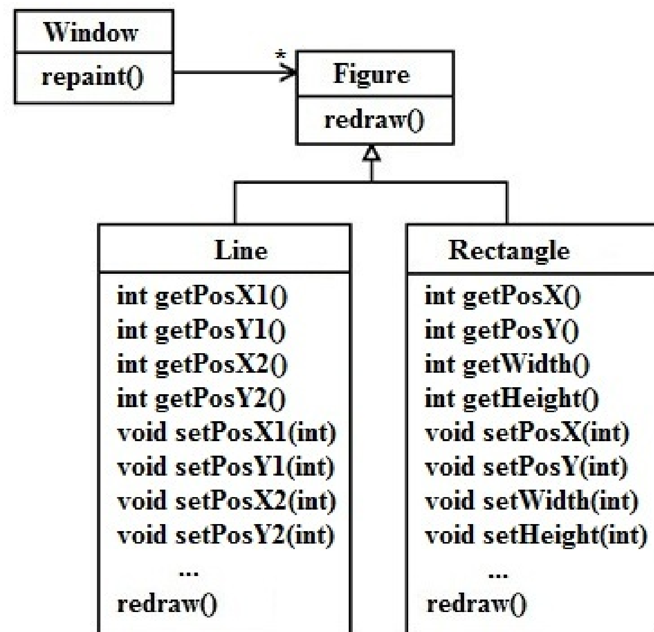

Figura 3.1. Diagrama de classes para o exemplo.

```
1  public void changeColorAll(Color c) {
2    Figure f;
3
4    for(Iterator iterator = figuresList.iterator(); iterator.hasNext(); ) {
5      f = (Figure)iterator.next();
6      f.setColor(c);
7    }
8
9    Editor.getWindowApp().repaint();
10 }
```

Figura 3.2. Função para alteração de cor com a chamada a *repaint*.

Em Chiba & Nakagawa [2004], os autores abordam esse problema e apresentam uma solução para uma linguagem específica, Josh. Josh dispõe do designador *updater*, que é definido pelo usuário, para selecionar as chamadas de método especificado pelo algoritmo baseado na dependência entre classes. Assim, podemos selecionar as chamadas para o método *repaint* e resolver o problema da modularização.

Para o propósito deste trabalho, nos interessa apenas os métodos que contêm chamadas a métodos *set**. A Figura 3.2 apresenta um método para alteração da cor de todas as figuras. Uma chamada ao método *setColor* é feita dentro de *changeColorAll* para cada figura presente na lista de figuras *figuresList* e, em seguida, é chamado *repaint*, para atualizar a tela da aplicação. Para não precisar repetir a chamada a *repaint* em todos os

```
1 pointcut callset() : call(* Figure+.set*(..));
2
3 pointcut alter() : contains(callset());
4
5 after() : alter() {
6    Editor.getWindowApp().repaint();
7 }
```

Figura 3.3. Solução elegante para o problema apresentado.

métodos que alteram atributos de *Figure*, semelhantemente ao método acima demonstrado, é desejado usar um aspecto para fazer essa inserção de forma automática. Seria necessário implementar em AspectJ um designador de *pointcut* que permita verificar as chamadas para os métodos *set* dentro de outros métodos, capturar o contexto destes e inserir uma chamada para *repaint* no final do seu código.

Na extensão *eaj* (***E****xtensible* ***A****spect****J***), criada pelo grupo desenvolvedor do *abc*, foi implementado um designador de *pointcut* que captura o contexto de pontos de junção que contêm um outro ponto de junção determinado como parâmetro. A esse designador foi dado o nome de *contains*. Usando essa extensão, é possível resolver o problema discutido acima. Com o trecho apresentado na Figura 3.3 inserido dentro de um aspecto, conseguimos resolver o problema em uma solução elegante.

O *pointcut callset* captura as chamadas para os métodos *set* (*setPosX*, *setColor*, etc) de elementos *Figure* e suas subclasses (*Line*, *Rectangle*, etc). O *pointcut alter* captura através do designador *contains* todos os pontos de junção em que *callset* atua, ou seja, todos os pontos do programa que contêm chamadas para os métodos *set**. Assim, *contains* consegue capturar o método *changeColorAll*, por exemplo, e inserir nele uma chamada para *repaint* através de um adendo *after*. Neste exemplo, o método *getWindowApp* da classe *Editor* retorna um *singleton* que representa a janela da aplicação (classe *Window*), que contém o método *repaint*.

O código da Figura 3.2 sem a chamada para *repaint*, juntamente com o código de aspecto apresentado na Figura 3.3, foram compilados usando o *abc* e a extensão *eaj*. A chamada para *repaint* deve ser inserida pelo aspecto. O resultado pode ser observado na Figura 3.4. O *abc* gera código objeto em *bytecode* e a Figura 3.4 apresenta um código java obtido usando-se o decompilador DAVA [Miecznikowski & Hendren, 2002] sobre esse *bytecode* gerado na compilação. Note que as chamadas ao método *repaint* foram corretamente inseridas.

Na seção seguinte, utilizamos o designador *contains* como exemplo. Apresentamos uma classe de sintaxe em XAJ para definir o designador *contains* e fazer com que ele funcione da forma como foi proposta. Para isso, além dos mecanismos já implementados,

```
1  public void changeColorAll(Color r1) throws java.lang.Throwable
2  {
3    Object n0;
4    Iterator r3;
5    n0 = null;
6
7    try
8    {
9      r3 = figuresList.iterator();
10
11     while (r3.hasNext())
12     {
13       ((Figure) r3.next()).setColor(r1);
14     }
15   }
16   catch (Throwable $r6)
17   {
18     if (n0 == null)
19     {
20       InsertRepaint.aspectOf();
21     }
22
23     Editor.getWindowApp().repaint();
24     throw $r6;
25   }
26
27   InsertRepaint.aspectOf();
28   Editor.getWindowApp().repaint();
29 }
```

Figura 3.4. Código decompilado por DAVA.

também precisamos descrever códigos a serem executados em tempo de costura. A definição eficiente de um designador de *pointcut* como o *contains* deve permitir que todo o código seja executado em tempo de costura, pois se trata de um designador não-dinâmico e só depende das informações estáticas do programa, não gerando código para fazer avaliações que usam informações dinâmicas, ou seja, obtidas em tempo de execução. Essa abordagem visa ter alta modularidade e menor complexidade, se comparada a *eaj*.

3.2 Solução em XAJ para o Problema Apresentado

Vimos que *eaj* implementa efetivamente o designador *contains*. Porém, o código que o torna funcional é complexo e está espalhado em vários arquivos do compilador *abc*. Usamos a extensibilidade de linguagem, em particular, XAJ, para resolver o problema da modularização do requisito transversal de atualização de tela, com a definição de designadores de *pointcut* pelo próprio usuário, permitindo, assim, automatizar o processo de extensão. Através da definição de uma classe de sintaxe (*syntaxclass*), podemos encapsular a especificação de uma extensão e criar o designador *contains* de forma automática,

```
1  public syntaxclass SC_Contains extends Pointcut_c {
2    @grammar extends basic_pointcut_expr {
3      SC_Contains ->
4      "contains" "(" p = pointcut_expr ")"
5      {:
6        SC_Contains x =
7              new SC_Contains(parser.pos(p, p), p);
8        RESULT = x;
9      :}
10     ;
11   }
12
13   // Definição do bloco OnWeaving (Figura 3.7)
14   ...
15
16   // Definição da AST
17   private abc.aspectj.ast.Pointcut param;
18   ...
19   public abc.weaving.aspectinfo.Pointcut
20                                      makeAIPointcut() {
21     return new AspectInfoContains(position,
22                               param.makeAIPointcut());
23   }
24   ...
25 }
```

Figura 3.5. Classe de sintaxe para o *contains*.

gerando o símbolo (*token*), a gramática, a AST e outras classes necessárias ao compilador de XAJ. Mas, primeiramente, tivemos que modificá-lo de maneira a aceitar que a classe de sintaxe suporte a definição de código a ser gerado em tempo de costura. Os detalhes das modificações feitas no compilador de XAJ são apresentados no Capítulo 4.

A classe de sintaxe que determina a extensão *contains* está representada na Figura 3.5, com algumas simplificações.

A linha 1 da Figura 3.5 indica a criação de uma classe de sintaxe em XAJ, de nome *SC_Contains*. Este código gera um arquivo *.java* com o mesmo nome. A classe gerada herda métodos e atributos de *abc.aspectj.ast.Pointcut_c*, para a definição de designadores de *pointcut*.

O código compreendido entre as linhas 2 e 11, contendo a declaração @grammar, gera as regras gramaticais e o *token* para o novo designador pretendido. Este tipo de geração automática estende o analisador léxico e a gramática da linguagem AspectJ, dispensando o trabalho de modificar vários arquivos do compilador à mão. Na gramática, a declaração @grammar cria um símbolo não-terminal e define a sintaxe para o mesmo. O resultado, gerado automaticamente a partir da classe de sintaxe, pode ser visto na Figura 3.6. Este mecanismo usa o módulo PPG [Brukman & Myers, 2008] do Polyglot [Nystrom et al., 2003], um *framework* utilizado no *frontend* do *abc*, para manipular produções gramaticais, importando definições de modo a possibilitar a geração de um novo

```
1  //--------------------Non Terminals-------------------
2  non terminal SC_Contains;
3
4  //--------------- syntaxclass SC_Contains
5
6  extend basic_pointcut_expr ::=
7    SC_Contains: rt
8    {:
9      RESULT = rt;
10   :};
11
12 SC_Contains ::=
13   CONTAINS LPAREN pointcut_expr :p RPAREN
14   {:
15     SC_Contains x =
16       new SC_Contains(parser.pos(p, p), p);
17     RESULT = x;
18   :};
```

Figura 3.6. Gramática para o *contains*.

não-terminal que estende um outro já existente.

As linhas de 16 ao final da Figura 3.5 representam o restante da definição da AST, já que uma *syntaxclass* também representa um nó da árvore de sintaxe abstrata.

A declaração @onWeaving, na Figura 3.7, representa a contribuição deste trabalho para a linguagem XAJ. Este trecho de código é responsável pela geração automática de um *AspectInfo*, uma estrutura utilizada no processo de costura, e que contém todas as informações necessárias para realizar tal tarefa. Classes que representam tal estrutura, por obrigação, têm como superclasse *abc.weaving.aspectinfo.Pointcut*, um *AspectInfo* geral. Logo, a classe gerada *AspectInfoContains* possui uma cláusula *extends* voltada a herança dos atributos e métodos de *abc.weaving.aspectinfo.Pointcut*.

O código principal que define o funcionamento da extensão *contains* está englobado nos métodos *matchesAt* e no método auxiliar *doShadows*, utilizado por este. Levando em conta a Figura 3.2 (sem a chamada para *repaint*) e a Figura 3.3, ao chamar *matchesAt* de *AspectInfoContains* (chamado ao encontrar o designador *contains*) o *MatchingContext* equivale a informações sobre o método *changeColorAll*, cuja *ShadowMatch* é compatível (*instanceof*) com *BodyShadowMatch*. Toda a sequência de comandos dentro de *changeColorAll* é capturada em *stmtsChain*. Sobre seus elementos são feitas operações para averiguar se o *pointcut* deve casar. Pelo exemplo, a situação que seria correspondida pelo *contains* é quando *current*, o elemento da iteração sobre *stmtsChain*, for a chamada para *setColor*. O método auxiliar *doShadows* recebe o contexto e a posição da chamada ao método *setColor* em *changeColorAll*. A partir de todos os *joinpoint shadows* do programa (todo ponto de junção potencial, pontos do programa que poderiam ser correspondidos por um *pointcut*), itera sobre todas as *ShadowType* encontradas pelo código, e chama

```
@onWeaving AspectInfoContains {
  Pointcut param = null;
  ...
  public Residue matchesAt(MatchingContext mc)
          throws SemanticException {
    Residue ret = NeverMatch.v();

    if (!(mc.getShadowMatch() instanceof BodyShadowMatch)) {
      return NeverMatch.v();
    }

    Chain stmtsChain = mc.getSootMethod().
                          getActiveBody().getUnits();

    for (current = (Stmt) stmtsChain.getFirst();
          current != null; current =
            (Stmt) stmtsChain.getSuccOf(current)) {
      StmtMethodPosition pos =
            new StmtMethodPosition(mc.getSootMethod(),
                                         current);

      if (doShadows(mc, pos) == AlwaysMatch.v()) {
        ret = AlwaysMatch.v();
        break;
      }
    }
    return ret;
  }

  public Residue doShadows(MatchingContext mc,
       MethodPosition pos)
          throws SemanticException {
    Residue ret = NeverMatch.v();

    for (Iterator i = abc.main.Main.v().
                     getAbcExtension().shadowTypes();
             i.hasNext();) {

      ShadowType st = (ShadowType) i.next();
      ShadowMatch sm;

      try {
        sm = st.matchesAt(pos);
      } catch (Throwable e) {
        ...
      }

      Residue currMatch = param.matchesAt(
        new MatchingContext(mc.getWeavingEnv(),
          mc.getSootClass(), mc.getSootMethod(), sm));

      if (currMatch == AlwaysMatch.v()) {
        ret = currMatch;
        break;
      }
    }
    ...
    return ret;
  }
}
```

Figura 3.7. Bloco *OnWeaving* para o *contains*.

matchesAt de cada uma delas para verificar se a *shadow* ocorre naquela posição e capturar seu *ShadowMatch*. Ao chamar *matchesAt* de *param*, o *MethodCall* **call(*(*).set*(..))**, passando um novo *MatchingContext* criado com atributos do contexto atual (*changeColorAll*) e a *ShadowMatch* de *setColor*, será verificada uma correspondência, ou seja, será retornado um resíduo que sempre é satisfeito (*AlwaysMatch*). Resumindo, isto verifica se o *pointcut* passado como parâmetro para *contains* se encontra em *changeColorAll*. É importante ressaltar que tal avaliação não gera resíduos a serem avaliados em tempo de execução, sendo *AlwaysMatch* e *NeverMatch* os únicos valores retornados, significando que o *pointcut contains* deve atuar no primeiro, e que não deve, no segundo.

A solução demonstrada nesta seção não condiz com a apresentada em Di Iorio et al. [2009], introduzindo alterações no mecanismo proposto anteriormente. A explicação para isto será relatada na Seção 6.2.

Capítulo 4

Detalhes de Implementação

Neste capítulo, são apresentados os procedimentos realizados para implementar um novo mecanismo em XAJ, de modo a atender aos objetivos deste projeto. Tais procedimentos estendem a linguagem XAJ, de modo a permitir o uso da declaração @onWeaving na classe de sintaxe. A primeira seção faz um apanhado geral sobre a melhor forma escolhida para implementar o mecanismo para gerar código automático em tempo de costura para extensões de designadores de *pointcut* estáticos, definidas pelo usuário. A segunda seção é responsável por apresentar as construções feitas para criar uma extensão responsável por representar a declaração @onWeaving na linguagem XAJ. A Seção 4.3, descreve as alterações feitas em XAJ para que essa construção pudesse ser usada e manipulada, de modo a permitir a definição de *pointcuts* em tempo de costura, usando o código especificado em @onWeaving.

4.1 Implementação de Mecanismo Automático Para Gerar Código Para Novos Pointcuts

As restrições impostas aos mecanismos de extensão de linguagens orientadas a aspectos limitam seu poder de expressividade, continuando com os problemas atribuídos a AspectJ, como a fragilidade de *pointcuts* [Störzer & Koppen, 2004].

A melhor maneira encontrada para estender o processo de costura de XAJ foi continuar com a mesma abordagem feita para @grammar, ou seja, criar uma anotação interna reconhecida pela linguagem. A *syntaxclass* foi modificada de maneira a permitir a definição de código responsável pelo processo de costura, adicionando a declaração @onWeaving, a qual irá encapsular o código que realiza tal processo. Essa anotação, implementada estendendo a linguagem XAJ segundo o modelo apresentado na Figura 4.1, se

```
1 [MODIFICADOR] syntaxclass IDENTIFICADOR [SUPER] [INTERFACES]
2 "{"
3   [@numberOfPasses INTEIRO PT_VIRG]
4   [@grammar (USING|EXTENDS|OVERRIDES) GRAMMAR_PRODUCTIONS]
5   [@onWeaving IDENTIFICADOR ONWEAVING_PRODUCTIONS]
6   [CLASS_BODY_DECLARATIONS]
7 "}"
```

Figura 4.1. Novo esquema para uma *syntaxclass*.

dá em bloco e aceita declarações de atributos, métodos e construtores, de modo a permitir o uso da construção apresentada na Seção 3.2. A primeira etapa da modificação se baseia na extensão do *lexer* do *abc*, fazendo com que ele reconheça o novo *token* @onWeaving relacionando o mesmo ao símbolo ONWEAVING na tabela de símbolos. A segunda etapa modifica o analisador sintático do compilador de XAJ adicionando as novas produções gramaticais relacionadas ao *token* @onWeaving na gramática da linguagem XAJ. Deve ser lembrado que a gramática de XAJ estende a gramática de AspectJ. A atribuição de um novo passo ao compilador também foi necessária, para manipular o bloco @onWeaving da *syntaxclass*.

Em seguida, precisamos criar a estrutura para armazenar as informações da nova construção, o nó da AST. Demos o nome de *OnWeaving_c* a tal estrutura; ela tem como superclasse a classe *polyglot.ext.jl.ast.Node_c* que representa nós da AST e está definida no *abc*. Além do mais, ela implementa uma interface *OnWeaving*. XAJ, assim como o *abc*, é baseado no padrão de projeto *Visitor*, dentre outros, que manipula estruturas sem que seja necessário alterá-las diretamente. A linguagem se baseia na programação orientada a interfaces a fim de evitar acoplamento desnecessário, melhorando o reuso. Para a classe *OnWeaving_c*, criamos um *Visitor* específico. O novo visitor *OnWeavingVisitor* estende *XAJVisitor*, o visitor para as classes visitáveis criadas por XAJ, e adiciona operações para manipular uma estrutura que implementa *OnWeaving*. Seu método *visitOnWeaving* recebe um elemento que implementa a interface *OnWeaving* e cria automaticamente a estrutura utilizada no processo de costura (*AspectInfo*) com as seguintes etapas:

- Criar arquivo *.java* com o nome especificado no símbolo **IDENTIFICADOR**;
- Criar uma classe com o modificador *public* cujo nome está especificado no símbolo **IDENTIFICADOR** e que estenda a superclasse *abc.weaving.aspectinfo.Pointcut*;
- Manipular **ONWEAVING_PRODUCTIONS**, sendo

  ONWEAVING_PRODUCTIONS = "{"CLASS_BODY_DECLARATIONS "}"

uma lista de *ClassMember*, que representam membros que podem ser declarados dentro de corpos de classes.

Com isso, conseguimos disponibilizar ao usuário um mecanismo que o permita estender a linguagem, criando suas próprias definições para novos designadores de *pointcut*, sem ter que, necessariamente, modificar à mão o código de um compilador existente, o que seria extremamente complicado. Consequentemente, ele também pode criar a extensão que resolve o problema citado na Seção 3.1, construindo um designador de *pointcut* semelhante ao *contains* da mesma forma que a descrita na Seção 3.2.

4.2 Criando uma Extensão para a Declaração @onWeaving

4.2.1 Estendendo o Analisador Léxico

O reconhecimento de novos símbolos (*tokens*) é feito através da adição de novas palavras-chave ao gerador de analisador léxico JFlex, uma implementação em Java baseada no JLex [Berk, 1997] e compatível com o CUP [Hudson, 1999], o analisador sintático. Para a extensão que representará a declaração @onWeaving, o reconhecimento do novo *token* (@onWeaving) no *lexer* do *abc* é conseguido com a modificação do arquivo de entrada para o JFlex, o arquivo *syntaxclass.flex*, que se encontra no pacote *xaj.scc.parse*, adicionando a seguinte linha:

"@onWeaving"{ return new OnWeavingToken(pos(), sym.ONWEAVING); }

A inserção deste trecho gera uma modificação na classe *Lexer_c*, o analisador léxico para o *abc*, e na classe *sym*, que representa a tabela de símbolos para o compilador, ou seja, contém os símbolos terminais ou *tokens* da gramática da linguagem. Ambas as classes também estão presentes no pacote *xaj.scc.parse*. Uma associação da nova palavra @onWeaving ao símbolo *ONWEAVING* é feita por meio de um código numérico, usado em um *case* no analisador léxico, para identificar qual estrutura construir.

A classe *OnWeavingToken* foi criada como subclasse de *polyglot.lex.Token*, a classe do compilador responsável por gerir todos os *tokens* da linguagem, e é responsável por criar referências especificamente para o *token* que representa a declaração @onWeaving. A classe tem um construtor que recebe como parâmetros a posição em que foi encontrado e o respectivo símbolo relacionado na tabela de símbolos, além de um método que

```
1 public class OnWeavingToken extends Token {
2   public OnWeavingToken(Position position, int symbol) {
3     super(position, symbol);
4   }
5
6   public String toString() {
7     return "@onWeaving";
8   }
9 }
```

Figura 4.2. Extensão do *lexer*.

```
1  /* add the possiblility of declaring a syntax class to type_declaration */
2  extend type_declaration ::=   // class | interface | aspect | syntaxclass
3       syntaxclass_declaration:s
4       {:
5         RESULT = s;
6       :}
7     ;
8
9  /* add the possiblility of declaring a syntaclass as class member */
10 extend class_member_declaration ::=
11      syntaxclass_declaration:s
12      {:
13        Grm.parserTrace("a␣class␣member␣declaration␣is␣a␣syntax␣class");
14        List l = new TypedList(new LinkedList(), ClassMember.class, false);
15        l.add(s);
16        RESULT = l;
17      :}
18    ;
```

Figura 4.3. Extensão para suportar a definição de uma *syntaxclass*.

retorna uma conversão do elemento dessa classe para uma *String*, como demonstrado na Figura 4.2.

4.2.2 Estendendo o Analisador Sintático

O analisador sintático ou *parser* do *abc* é baseado no *framework* Polyglot. Podemos estender de forma fácil e segura as alternativas para não-terminais existentes na gramática da linguagem base usando o módulo PPG, presente neste *framework*. A palavra-chave *extend* nos permite isto. Os não-terminais *type_declaration* e *class_member_declaration* da gramática de Java foram estendidos para suportar a construção *syntaxclass*, conforme a Figura 4.3. Dessa forma, uma declaração *syntaxclass* pode aparecer tanto como um tipo de declaração para classes Java, semelhante a declarações de interface e classes usuais, quanto em corpos de classes Java, assim como declarações de métodos e variáveis, respectivamente, na linguagem XAJ.

Para implementar a declaração @onWeaving segundo o modelo apresentado na

Figura 4.1, a definição do não-terminal *syntaxclass_body* contido na definição do não-terminal *syntaxclass_declaration* foi alterada de

... syntax_declaration:sd class_body_declarations_opt:l ...

para

... syntax_declaration:sd on_weaving_opt:ow class_body_declarations_opt:l ...

de modo a permitir construções do tipo

@onWeaving IDENTIFIER { onweaving_productions }

dentro do corpo da syntaxclass. A Figura 4.4 mostra o fragmento mais significativo da gramática para suportar tal extensão. A gramática completa pode ser visualizada no Apêndice A.

Este código adiciona a produção do símbolo @onWeaving às regras já existentes para o símbolo não-terminal *syntaxclass_body*. Note que o sufixo *opt* (de *optional*, sendo uma simples convenção de nomenclatura) em *on_weaving_opt*, indica que pode ou não haver uma declaração desse tipo para uma construção *syntaxclass*, apresentando produção gerando *lambda* em caso negativo, conforme consta na Figura 4.4. Como está expresso por este exemplo, símbolos terminais são indicados por letras maiúsculas. É possível vincular o resultado da análise de cada símbolo gramatical a um identificador, indicado por um dois pontos e um nome. Por exemplo, ligamos o resultado do reconhecimento do símbolo *ONWEAVING* para *o*, e o resultado de *on_weaving_declaration_body* para *productions*. Estes resultados nomeados podem então ser usados na ação do analisador associada a uma produção. Esta ação é denominada ação semântica e está delimitada entre chaves e dois pontos. Aqui usamos os resultados do primeiro e último símbolo no lado direito da produção para calcular a posição (através da chamada *parser.pos(o, productions)*). Posições no Polyglot são sempre uma localização de início (arquivo de origem, número da linha, número da coluna) juntamente com uma de final. Por toda parte do *abc* é tomado um grande cuidado para preservar tais informações de posição, de modo que seja possível localizar a origem de todo fragmento de código, mesmo após otimizações terem sido aplicadas.

4.2.3 Adicionando um Novo Nó AST

Reconhecido o novo símbolo e suas regras de sintaxe, precisamos de uma estrutura de dados para fornecer as operações necessárias e guardar algumas informações a respeito

```
1  terminal Token ONWEAVING;
2
3  non terminal OnWeaving on_weaving_opt;
4  non terminal List/*<ClassMember>*/ on_weaving_declaration_body;
5
6  ...
7
8  syntaxclass_body ::=
9       LBRACE:lb numberofpasses_opt:n syntax_declaration:sd
10      on_weaving_opt:ow class_body_declarations_opt:l RBRACE:rb
11      {:
12        Grm.parserTrace("syntax␣class␣body");
13        RESULT = parser.nf.SyntaxClassBody(parser.pos(lb,rb), n, sd, ow, l);
14      :}
15    ;
16
17 on_weaving_opt ::=
18      ONWEAVING:o IDENTIFIER:id on_weaving_declaration_body:productions
19      {:
20        Grm.parserTrace("onWeaving␣declaration");
21        RESULT = parser.nf.OnWeaving(parser.pos(o, productions),
22                                     id.getIdentifier(), productions);
23      :}
24     |
25      {:
26        Grm.parserTrace("There␣isn't␣onWeaving␣declaration");
27        RESULT = parser.nf.OnWeaving(null, "", null);
28      :}
29    ;
30
31 on_weaving_declaration_body ::=
32      LBRACE class_body_declarations_opt:l RBRACE
33      {:
34        RESULT = l;
35      :}
36    ;
```

Figura 4.4. Extensão para suportar a construção @onWeaving.

da declaração @onWeaving, como a posição em que foi encontrado, o identificador que irá se referir a classe gerada automaticamente e as produções que irão compor o corpo dessa classe. Na gramática, quando é encontrada uma construção do tipo @onWeaving, um novo nó na AST deve ser associado ao resultado da produção gramatical. Denominada *OnWeaving_c*, esta classe AST é subclasse de *polyglot.ext.jl.ast.Node_c* (herdando seus métodos e atributos) e implementa os métodos de uma interface *OnWeaving*. Seu construtor tem como parâmetros a posição, o identificador e as produções, conforme demonstrado pela chamada ao método fábrica no trecho a seguir:

parser.nf.OnWeaving(parser.pos(o, productions), id.getIdentifier(), productions);

Detalhando o trecho acima temos que:

- *parser.nf.OnWeaving(...)* constrói um objeto AST *OnWeaving_c* usando o padrão

```
1 public interface OnWeaving extends Node, Visitable {
2   public Position getPos();
3   public String getName();
4   public List<ClassMember> getProductions();
5 }
```

Figura 4.5. Interface para a classe AST da construção @onWeaving.

```
1 public interface Visitable {
2   void accept(Visitor v);
3 }
```

Figura 4.6. Interface para as classes visitáveis.

de projeto Fábrica, onde *nf* significa *Node Factory*;

- *parser.pos(o, productions)* cria um objeto *Position* que contém dados da posição em que foi encontrada a construção (nome do arquivo, linha, coluna, etc) e será passado para a superclasse *Node_c* e em seguida ao *Polyglot*;
- *id.getIdentifier()* pega o nome do identificador armazenado em *id* sendo passado para o atributo *name* de *OnWeaving_c*;
- *productions* é uma lista de declarações representando corpos de classes Java (*ClassMember*) passado para o atributo *productions* de *OnWeaving_c*.

A interface *OnWeaving* é mostrada na Figura 4.5. Chamamos a atenção para as interfaces que ela implementa. A classe AST está representada na Figura 4.7. Note que ela possui um método *accept*, obedecendo ao padrão de projeto *Visitor*. Esse método é de implementação obrigatória pela implementação da interface *xaj.scc.ast.Visitable* mostrada na Figura 4.6, que representa as classes visitáveis pelo padrão de projeto *Visitor*. Maiores detalhes a respeito desse padrão de projeto e sua implementação no compilador de XAJ serão explicados mais adiante.

4.3 Adaptação do Sistema para Permitir a Definição de Pointcuts em Tempo de Costura

4.3.1 Criação de um Novo Visitor

A AST (do inglês, *Abstract Syntax Tree*) contém todas as informações necessárias sobre a nova construção. Essas informações devem ser manipuladas para obtermos o resultado

```
1  public class OnWeaving_c extends Node_c implements OnWeaving {
2    String name;
3    List<ClassMember> productions;
4
5    public OnWeaving_c(Position pos, String name,
6        List<ClassMember> productions) {
7      super(pos);
8      this.name = name;
9      this.productions = productions;
10   }
11
12   public Position getPos() {
13     return position;
14   }
15
16   public String getName() {
17     return name;
18   }
19
20   public List<ClassMember> getProductions() {
21     return productions;
22   }
23
24   public void prettyPrint(CodeWriter w, PrettyPrinter tr) {
25     w.newline(4);
26     w.begin(0);
27     w.write("@onWeaving␣" + name);
28     w.write("␣{");
29     w.newline(4);
30     w.begin(0);
31     for(ClassMember p : productions) {
32       w.begin(0);
33       p.prettyPrint(w, tr);
34       w.end();
35     }
36     w.end();
37     w.newline();
38     w.write("}");
39     w.end();
40   }
41
42   public void accept(Visitor v) {
43     v.visitOnWeaving(this);
44   }
45 }
```

Figura 4.7. Classe para representar a AST da construção @onWeaving.

que queremos, no caso, a geração automática do *AspectInfo* para as extensões definidas pelo usuário. O *Visitor* é um padrão de projeto que adiciona a um objeto novas operações a serem realizadas sobre o mesmo, sem modificar diretamente a estrutura base desse objeto. Assim, por exemplo, podemos calcular áreas e perímetros de figuras geométricas sem, necessariamente, termos que modificar esses elementos. O código para realizar essas operações se concentrará em uma classe *Visitor*, sendo criada, por efeito de modularização, uma classe para cada operação (para o exemplo, dois *Visitors* seriam criados). Dessa forma, equivalentemente ao exemplo dado, um elemento *Visitor* também se torna necessário para realizar as devidas operações para determinar a criação do *AspectInfo*, a partir das informação de *OnWeaving_c*. O *Visitor OnWeavingVisitor.java* foi criado para gerar automaticamente o *AspectInfo* para as extensões e seu código está detalhado na Figura 4.8.

Ao criar o *AspectInfo*, o *Visitor OnWeavingVisitor* procura no corpo da declaração @onWeaving da classe de sintaxe pelos métodos de implementação obrigatória a todo *AspectInfo*. São eles: *getFreeVars*, *inline*, *matchesAt*, *registerSetupAdvice* e *toString*, além do próprio construtor. Se algum deles não tiver ocorrência no arquivo de entrada, ele cuida de implementar automaticamente estes métodos obrigatórios, não definidos pelo programador. Isto faz com o que o programador possa se concentrar apenas na implementação dos métodos necessários para que a extensão funcione corretamente, já que alguns dos métodos obrigatórios necessitam apenas da implementação *default*.

4.3.2 Alterando a Sequência de Passos do Compilador

A sequência de operações que o compilador *abc* realiza é definida por passos, sendo cada uma implementada por um *Visitor*. Um dos passos do *abc*, por exemplo, é chamar o Polyglot sobre a AST, criando uma AST anotada com informações de tipo. Esse passo corresponde a um dos passos base (*PARSE*) realizados por este compilador. Ele é definido no arquivo *ExtensionInfo.java* presente no pacote *abc.aspectj*. A chamada para qualquer passo após o passo base deve ser realizada como a seguir:

afterPass(passes, Pass.PARSE, new VisitorPass(...));

Para chamar as operações do *Visitor OnWeavingVisitor* criado, precisamos dizer ao *abc* que mais passos serão necessários para realizar a compilação completa daquela extensão. Logo, modificamos a classe *abc.aspectj.ExtensionInfo* de maneira a informar ao compilador que haverá mais um passo a ser realizado e que esse passo irá executar nosso *Visitor* e criar o *AspectInfo*. As alterações para modificar a sequência de execução

```
1  public class OnWeavingVisitor extends XAJVisitor {
2    private static PrintStream output;
3    private static String source;
4    private List<String> listImplementedMethods = new ArrayList<String>();
5
6    public OnWeavingVisitor(Job job, TypeSystem typeSystem,
7        NodeFactory nodeFactory) {
8      super(job, typeSystem, nodeFactory);
9    }
10
11   public void visitOnWeaving(OnWeaving ow) {
12     if (getVisitType() == VisitType.ENTER) {
13       String fileName = "tmp/" + ((OnWeaving_c)ow).getName() + ".java";
14       String testClass = new String();
15
16       testClass ="/**␣This␣Code␣is␣automatically␣generated\n" +
17       "*\n*␣@author␣Cleverson␣Carneiro␣Trevenzoli\n" +
18        "*␣@author␣Vladimir␣Oliveira␣Di␣Iorio\n*/\n\n";
19
20       testClass += "public␣class␣" + ((OnWeaving_c)ow).getName() +
21                    "␣extends␣Pointcut␣{\n";
22
23       PrettyPrinter pp = new PrettyPrinter();
24       ByteArrayOutputStream ba = new ByteArrayOutputStream();
25       CodeWriter cd = new CodeWriter(ba,10);
26
27       for (ClassMember i: ((OnWeaving_c)ow).getProductions()) {
28         // obter a lista dos métodos implementados
29         if(i instanceof MethodDecl) {
30           listImplementedMethods.add(((MethodDecl)i).name());
31         }
32
33         pp.printAst(i, cd);
34         testClass += ba.toString();
35         ba.reset();
36         testClass += "\n";
37       }
38
39       // verificar se os métodos obrigatórios
40       // foram implementados pelo usuário
41       if(!listImplementedMethods.contains("m") {
42         // inserir a implementação default para "m" no arquivo
43         ...
44       }
45
46       testClass += "}";
47       FileWriter fw;
48       try {
49         fw = new FileWriter(fileName);
50         fw.write(testClass);
51         fw.close();
52       } catch (IOException e) {
53         e.printStackTrace();
54       }
55     }
56   }
57 }
```

Figura 4.8. Visitor para a construção @onWeaving.

```
1  public static final ID ONWEAVING_VISITOR =
2                                   new Pass.ID("Visitor-OnWeaving");
3  public static final ID FINISHED_ONWEAVING_VISITOR =
4                                   new Pass.ID("finished-Visitor-OnWeaving");
5
6  ...
7
8  // criando novos passos que executarão o visitor OnWeaving
9  afterPass(passes, Pass.PARSE,
10                   new VisitorPass(ONWEAVING_VISITOR, job,
11                   new OnWeavingVisitor(job, typeSystem(), nodeFactory())));
12 afterPass(passes, ONWEAVING_VISITOR,
13                   new GlobalBarrierPass(FINISHED_ONWEAVING_VISITOR, job));
```

Figura 4.9. Passos para chamar o Visitor.

dos passos do *abc* se encontram na Figura 4.9.

O comando *new Pass.ID(...)* cria um identificador para um novo passo, determinado pelo nome passado como parâmetro. É necessário criar tanto um passo para início de operação quanto um para sinalizar a finalização da mesma. O método *afterPass(...)*, realiza uma operação após a execução de um passo especificado, como sugere o nome. Na Figura 4.9, após o passo Pass.PARSE chamamos o passo ONWEAVING_VISITOR criando o objeto *OnWeavingVisitor*, iniciando a visitação da AST e realizando a operação para construir o *AspectInfo*. Consecutivamente, após o passo ONWEAVING_VISITOR, finalizamos a operação criando uma barreira de aviso ao compilador (comando *GlobalBarrierPass(...)*) com o passo de término FINISHED_ONWEAVING_VISITOR.

4.3.3 Alterações na Fábrica Abstrata

O compilador de XAJ também é baseado no padrão de projeto *Abstract Factory* ou Fábrica Abstrata, que é responsável por gerir a criação dos nós da AST, sendo chamados no *parser* da linguagem. Em *xaj.scc.ast.SCNodeFactory*, a interface para a fábrica, foi modificada a declaração para o construtor das estruturas que representam corpos de uma *syntaxclass*, devido ao fato de que, a partir deste trabalho, um corpo da classe de sintaxe pode conter uma declaração @onWeaving. Esse elemento foi adicionado ao construtor da AST que representa o corpo de uma classe de sintaxe. Ao arquivo também foi adicionada a declaração referente a criação do nó da árvore de sintaxe abstrata *OnWeaving*, referente a declaração @onWeaving. A Figura 4.10 mostra as modificações feitas.

A Figura 4.11 mostra as modificações feitas na fábrica concreta *xaj.scc.ast.SCNodeFactory_c* que implementa a interface *xaj.scc.ast.SCNodeFactory* e contém a implementação das declarações desta interface.

```
1  public interface SCNodeFactory extends AJNodeFactory {
2    ...
3
4    public SyntaxClassBody SyntaxClassBody(Position pos,
5         NumberOfPasses number, SyntaxDeclaration sd,
6         OnWeaving ow, List<ClassMember> classmembers);
7
8    public OnWeaving OnWeaving(Position pos, String name,
9                              List<ClassMember> productions);
10 }
```

Figura 4.10. Alteração e adição das declarações na interface *SCNodeFactory*.

```
1  public class SCNodeFactory_c extends AJNodeFactory_c
2                                    implements SCNodeFactory {
3    ...
4
5    public xaj.scc.ast.SyntaxClassBody SyntaxClassBody(Position pos,
6        xaj.scc.ast.NumberOfPasses number, SyntaxDeclaration sd,
7        OnWeaving ow, List<ClassMember> classmembers) {
8      SyntaxClassBody n = new SyntaxClassBody_c(pos, number, sd,
9                                        ow, classmembers);
10     n = (SyntaxClassBody)n.ext(((SCExtFactory)extFactory()).
11                                             extSyntaxClassBody());
12     n = (SyntaxClassBody)n.del(((SCDelFactory)delFactory()).
13                                             delSyntaxClassBody());
14     return n;
15   }
16
17   public xaj.scc.ast.OnWeaving OnWeaving(Position pos, String name,
18       List<ClassMember> productions) {
19     OnWeaving o = new OnWeaving_c(pos, name, productions);
20     return o;
21   }
22 }
```

Figura 4.11. Alteração e adição das implementações na classe *SCNodeFactory_c*.

4.3.4 Modificação da Classe de Sintaxe

A fábrica foi modificada para suportar a nova funcionalidade adicionada à classe de sintaxe. Para isso, modificações foram feitas diretamente na classe AST *SyntaxClassBody_c* presente em *xaj.scc.ast*, que representa o corpo de uma classe de sintaxe. Assim, foi adicionado um membro privado *onWeaving* do tipo interface *OnWeaving*. Modificamos o construtor que agora receberá como parâmetro um objeto *OnWeaving* e armazenará o mesmo em *onWeaving*. Métodos *get* e *set* para este novo membro da classe foram criados. Além disso, os métodos *reconstruct*, *visitChildren* e *prettyPrint* foram alterados para criar o membro *onWeaving* ao reconstruir o nó AST, visitar elementos *OnWeaving* e chamar o *prettyPrint* para a nova construção, respectivamente. A Figura 4.12 mostra as principais modificações realizadas na classe AST *xaj.scc.ast.SyntaxClassBody_c*.

```
1  public class SyntaxClassBody_c extends AJClassBody_c
2      implements SyntaxClassBody {
3    ...
4    private OnWeaving onWeaving;
5
6    public SyntaxClassBody_c(Position pos, NumberOfPasses n,
7        SyntaxDeclaration s, OnWeaving o, List members) {
8      super(pos, members);
9      numberOfPasses = n; syntaxDeclaration = s; onWeaving = o;
10   }
11
12   public OnWeaving getOnWeaving() { return onWeaving; }
13
14   public SyntaxClassBody setOnWeaving(OnWeaving ow) {
15     SyntaxClassBody_c n = (SyntaxClassBody_c) copy();
16     n.onWeaving = ow;
17     return n;
18   }
19
20   protected SyntaxClassBody_c reconstruct(NumberOfPasses n,
21       SyntaxDeclaration s, OnWeaving o, List members) {
22     if(this.numberOfPasses != n || this.syntaxDeclaration != s ||
23         this.onWeaving != o || !CollectionUtil.equals(members,
24                                                  this.members)) {
25       SyntaxClassBody_c node = (SyntaxClassBody_c) copy();
26       node.numberOfPasses = n;
27       node.syntaxDeclaration = s;
28       node.onWeaving = o;
29       node.members = TypedList.copyAndCheck(members,
30                                        ClassMember.class, true);
31       return node;
32     }
33     return this;
34   }
35
36   public Node visitChildren(NodeVisitor v) {
37     NumberOfPasses nop = (NumberOfPasses) this.numberOfPasses.visit(v);
38     SyntaxDeclaration sd =
39                 (SyntaxDeclaration) this.syntaxDeclaration.visit(v);
40     OnWeaving ow = (OnWeaving) this.onWeaving.visit(v);
41     List members = visitList(this.members, v);
42     Node n = reconstruct(nop, sd, ow, members);
43     return n;
44   }
45
46   public void prettyPrint(CodeWriter w, PrettyPrinter tr) {
47     w.newline(4);  w.begin(0);
48     numberOfPasses.prettyPrint(w, tr);
49     w.newline(0);
50     syntaxDeclaration.prettyPrint(w, tr);
51     w.newline(0);
52     onWeaving.prettyPrint(w, tr);
53     w.newline(0);  w.end();
54     for(Object o : members()) {
55       ClassMember m = (ClassMember) o;
56       w.newline();  m.prettyPrint(w, tr);
57     }
58   }
59   ...
60 }
```

Figura 4.12. Principais modificações para SyntaxClassBody_c.java.

```
1 public interface Visitor {
2   ...
3 
4   void visitOnWeaving(OnWeaving ow);
5 }
```

Figura 4.13. Adicionando método na interface Visitor.

```
1 public class XAJVisitor
2        extends ErrorHandlingVisitor implements Visitor {
3   ...
4 
5   @Override
6   public void visitOnWeaving(OnWeaving ow) {
7 
8   }
9 }
```

Figura 4.14. Implementando o método da interface Visitor.

4.3.5 Modificação do Visitor

O padrão de projeto *Visitor* implementado em XAJ é composto pelo *Visitor XAJVisitor* e pela interface *Visitor*, presentes no pacotes *xaj.scc.visit* e *xaj.scc.ast*, respectivamente. A classe *XAJVisitor* implementa a interface *Visitor*. Para disparar o visitor *OnWeavingVisitor* sobre declarações @onWeaving e criar o *AspectInfo*, as modificações mostradas nas Figuras 4.13 e 4.14 foram necessárias.

Note que o método *visitOnWeaving* de *XAJVisitor* contém uma implementação vazia para a funcionalidade, já que a avaliação em tempo de execução irá casar e chamar o visitor concreto correspondente (*OnWeavingVisitor*, para este caso).

Capítulo 5

Testes e Avaliação

Neste capítulo iremos avaliar os resultados obtidos com o desenvolvimento do mecanismo proposto, através de um exemplo de aplicação que utiliza a extensão *contains* criada pela classe de sintaxe de XAJ.

5.1 Validação dos Resultados

Para comprovar a validade do nosso trabalho, foi usado o decompilador DAVA [Miecznikowski & Hendren, 2002] sobre o *bytecode* gerado na compilação de um programa-exemplo usando a extensão *contains* construída pelo novo mecanismo @onWeaving de XAJ. Como explicado na Seção 3.1, o exemplo para motivação é uma aplicação de desenhos baseados em figuras geométricas. Para a validação dos resultados simulamos esta mesma aplicação construindo, para esse fim, as seguintes classes:

1. *Figure*: classe mãe para todas as figuras geométricas suportadas pelo editor;
2. *Line*: subclasse de *Figure* representando o elemento linha;
3. *Rectangle*: subclasse de *Figure* representando o elemento retângulo;
4. *Editor*: classe principal que contém o método *main* e inicia a execução da aplicação;
5. *Window*: classe responsável por gerir a janela do editor;
6. *InsertRepaint*: aspecto modificador responsável por modularizar o interesse transversal de redesenho de tela. A todo momento em que um atributo de *Figure* é modificado por um método *set*, a janela do editor deve ser atualizada.

```
 1 public class Figure {
 2   private int x, y;
 3   private Color c;
 4
 5   public Figure(int x, int y, Color c) {
 6     this.x = x;
 7     this.y = y;
 8     this.c = c;
 9   }
10
11   // getters e setters
12   ...
13
14   public void redraw() {
15     // código para redesenhar a figura
16     ...
17   }
18 }
```

Figura 5.1. Classe Figure.

A classe *Figure* demostrada na Figura 5.1 disponibiliza atributos e operações compartilhados por todas as figuras geométricas. Por motivo de simplicidade, o código para o método responsável por redesenhar a tela da aplicação foi omitido, pois o mesmo se torna irrelevante para a realização do teste, já que queremos apenas analisar o efeito do aspecto sobre o código e não obter um programa funcional. Nossa aplicação também se limitou a não ter muitas subclasses de *Figure*, tendo somente as classes Rectangle e Line (Figuras 5.3 e 5.2), seus métodos *set* e *get* para os devidos atributos e seus construtores.

A classe *Editor* (Figura 5.4) é a base de toda a aplicação. Contendo o método *main*, ela inicia a execução da aplicação (código omitido) e cria elementos figuras adicionando-os em *figuresList*, a lista de figuras. Note que a criação desses objetos dentro da função *main* foi somente realizada por motivos de testes, pois em uma aplicação real as figuras seriam criadas por meio de uma interface gráfica, através da qual um usuário poderia iteragir com o programa.

O *Editor* também possui o método *getWindowApp()* para acessar uma instância única para a janela da aplicação, através de *Window.getInstance()*.

A classe *Window* é implementada como um *Singleton*, padrão de projeto que garante a existência de apenas uma instância de uma classe, mantendo um ponto de acesso global ao seu objeto. Assim, garantimos que somente uma janela seja criada pelo editor e que as operações de atualização sejam realizadas sobre ela.

O aspecto *InsertRepaint* é responsável por inserir chamadas a *repaint* no contexto dos métodos que contém chamadas para métodos que modificam os atributos de figuras (métodos *set*). Sua construção está representada na Figura 5.6 e utiliza a nova extensão *contains*, criada pelo mecanismo proposto nesta dissertação. O adendo *after* é responsável

```
1  public class Line extends Figure {
2    private int x2, y2;
3
4    public Line(int posX1, int posX2, int posY1, int posY2, Color c) {
5      super(posX1, posY1, c);
6      this.x2 = posX2;
7      this.y2 = posY2;
8    }
9
10   public void setX2(int posX2) {
11     this.x2 = posX2;
12   }
13
14   public void setY2(int posY2) {
15     this.y2 = posY2;
16   }
17
18   public int getX2() {
19     return this.x2;
20   }
21
22   public int getY2() {
23     return this.y2;
24   }
25 }
```

Figura 5.2. Classe Line.

```
1  public class Rectangle extends Figure {
2    private int w, h;
3
4    public Rectangle(int posX, int posY, int w, int h, Color c) {
5      super(posX, posY, c);
6      this.w = w;
7      this.h = h;
8    }
9
10   public void setW(int w) {
11     this.w = w;
12   }
13
14   public void setH(int h) {
15     this.h = h;
16   }
17
18   public int getW() {
19     return this.w;
20   }
21
22   public int getH() {
23     return this.h;
24   }
25 }
```

Figura 5.3. Classe Rectangle.

```java
public class Editor {
  private static List figuresList;

  public static void changeColorAll(Color c) {
    Figure f;

    for(Iterator iterator = figuresList.iterator(); iterator.hasNext(); ) {
      f = (Figure)iterator.next();
      f.setColor(c);
    }
  }

  public static void moveFigure(Figure f, int distX, int distY) {
    if(f instanceof Line)
      moveLine((Line)f, distX, distY);
    else
    if(f instanceof Rectangle)
      moveRectangle((Rectangle)f, distX, distY);
  }

  public static void moveLine(Line l, int distX, int distY) {
    l.setX(l.getX() + distX);
    l.setY(l.getY() + distY);
    l.setX2(l.getX2() + distX);
    l.setY2(l.getY2() + distY);
  }

  public static void moveRectangle(Rectangle r, int distX, int distY) {
    r.setX(r.getX() + distX);
    r.setY(r.getY() + distY);
  }

  public static void main(String args[]) {
    // código para iniciar a aplicação
    ...

    figuresList = new ArrayList();

    Line l = new Line(2, 4, 8, 12, new Color(255, 0, 0));
    figuresList.add(l);

    Rectangle r = new Rectangle(2, 4, 8, 8, new Color(0, 0, 255));
    figuresList.add(r);

    changeColorAll(new Color(255, 0, 0));

    moveFigure(r, 10, -20);
  }

  public static Window getWindowApp() {
    return Window.getInstance();
  }
}
```

Figura 5.4. Classe Editor.

```
1  public class Window {
2    // instância estática privada que será acessada
3    private static final Window INSTANCE = new Window();
4
5    // construtor privado obedecendo ao padrão Singleton
6    private Window() {
7      // operações de inicialização da classe
8      ...
9    }
10
11   // método que irá retornar uma instância única do objeto
12   public static synchronized Window getInstance() {
13     return INSTANCE;
14   }
15
16   public void repaint() {
17     // código para atualizar a janela da aplicação
18     // chamando redraw() para cada elemento Figure
19     ...
20   }
21 }
```

Figura 5.5. Classe Window.

```
1 public aspect InsertRepaint {
2   pointcut callset() : call(* Figure+.set*(..));
3
4   pointcut alter() : contains(callset());
5
6   after() : alter() {
7     Editor.getWindowApp().repaint();
8   }
9 }
```

Figura 5.6. Aspecto InsertRepaint.

por inserir código objeto responsável por capturar a janela do editor e chamar seu método *repaint*, que atuará conforme foi explicado.

O resultado esperado, conforme a configuração do nosso programa-exemplo, é que nos métodos *changeColorAll*, *moveLine* e *moveRectangle* da classe *Editor*, haja a inserção da chamada a *repaint* após as alterações dos atributos de *Figure*. Isso pode ser constatado pelas Figuras 5.7 e 5.8, que representam o código decompilado da classe *Editor* da aplicação-exemplo. Note que a inserção do código do aspecto foi feita corretamente conforme esperado (linhas 21, 25, 58 e 62 da primeira parte e linhas 14 e 18 da segunda).

Não obstante, quaisquer outros métodos, além dos demonstrados, que também modificam atributos de elementos figura podem ser capturados e ocorrer a inserção automática pelo aspecto da chamada para *repaint*, ao final de sua execução. Este caso é garantido pelo código do aspecto, da forma como está implementado na Figura 5.6.

```
 1 public class Editor
 2 {
 3   private static List figuresList;
 4
 5   public static void changeColorAll(Color  r1) throws java.lang.Throwable
 6   {
 7     Object n0;
 8     Iterator r3;
 9     n0 = null;
10     try
11     {
12       r3 = figuresList.iterator();
13       while (r3.hasNext())
14       {
15         ((Figure) r3.next()).setColor(r1);
16       }
17     }
18     catch (Throwable $r6)
19     {
20       if (n0 == null) { InsertRepaint.aspectOf(); }
21       Editor.getWindowApp().repaint();
22       throw $r6;
23     }
24     InsertRepaint.aspectOf();
25     Editor.getWindowApp().repaint();
26   }
27
28   public static void moveFigure(Figure  r1, int  i0, int  i1)
29   {
30     if ( ! (r1 instanceof Line))
31     {
32       if (r1 instanceof Rectangle)
33       {
34         this.moveRectangle((Rectangle) r1, i0, i1);
35       }
36     }
37     else
38     {
39       this.moveLine((Line) r1, i0, i1);
40     }
41   }
42
43   public static void moveLine(Line  r1, int  i0, int  i1)
44                                                  throws java.lang.Throwable
45   {
46     Object n0;
47     n0 = null;
48     try
49     {
50       r1.setX(r1.getX() + i0);
51       r1.setY(r1.getY() + i1);
52       r1.setX2(r1.getX2() + i0);
53       r1.setY2(r1.getY2() + i1);
54     }
55     catch (Throwable $r2)
56     {
57       if (n0 == null) { InsertRepaint.aspectOf(); }
58       Editor.getWindowApp().repaint();
59       throw $r2;
60     }
61     InsertRepaint.aspectOf();
62     Editor.getWindowApp().repaint();
63   }
```

Figura 5.7. Classe Editor decompilada por DAVA (Parte 1).

```
1   public static void moveRectangle(Rectangle  r1, int  i0, int  i1)
2                                            throws java.lang.Throwable
3   {
4     Object n0;
5     n0 = null;
6     try
7     {
8       r1.setX(r1.getX() + i0);
9       r1.setY(r1.getY() + i1);
10    }
11    catch (Throwable $r2)
12    {
13      if (n0 == null) { InsertRepaint.aspectOf(); }
14      Editor.getWindowApp().repaint();
15      throw $r2;
16    }
17    InsertRepaint.aspectOf();
18    Editor.getWindowApp().repaint();
19  }
20
21  public static void main(String[]  r0)
22  {
23    Color r2, r5;
24    Line r3;
25    Rectangle r6;
26    figuresList = new ArrayList();
27    r2 = new Color(255, 0, 0);
28    r3 = new Line(2, 4, 8, 12, r2);
29    figuresList.add(r3);
30    r5 = new Color(0, 0, 255);
31    r6 = new Rectangle(2, 4, 8, 8, r5);
32    figuresList.add(r6);
33    Editor.changeColorAll(new Color(255, 0, 0));
34    Editor.moveFigure(r6, 10, -20);
35  }
36
37  public static Window getWindowApp()
38  {
39    return Window.getInstance();
40  }
41
42  static
43  {
44
45  }
46 }
```

Figura 5.8. Classe Editor decompilada por DAVA (Parte 2).

```
1  public static void moveFigureAll(int distX, int distY) {
2    Figure f;
3
4    for(Iterator iterator = lista.iterator(); iterator.hasNext(); ) {
5      f = (Figure)iterator.next();
6      moveFigure(f, distX, distY);
7    }
8  }
```

Figura 5.9. Método *moveFigureAll*.

Ao analisar o código apresentado, podemos inferir que acabamos por demonstrar um exemplo de método em que o designador *contains* de *eaj* não funciona exatamente como desejado. O ideal seria inserir uma única chamada a *repaint* no final do método *moveFigure*, de forma semelhante ao que foi feito com *changeColorAll*, ao invés de inserir uma chamada em cada um dos métodos *moveLine* e *moveRectangle*, já que estes são chamados a partir de uma chamada anterior a *moveFigure*. A extensão *contains* conforme foi utilizada no aspecto prevê somente efeitos sobre os métodos *set** que aparecem dentro dos outros métodos chamados a partir de *moveFigure* e não no próprio método. O exemplo funcionou porque o método *moveFigure* chama apenas um método *move** de cada vez, não fazendo chamadas desnecessárias a *repaint*. Agora suponha o método *moveFigureAll*, demonstrado na Figura 5.9, que percorre toda a lista de figuras. Ele irá fazer várias chamadas a *repaint*, para cada figura movida, em vez de uma única chamada, no final do método mais externo.

A captura de *pointcuts* nem sempre funciona como o esperado. Como uma possível alternativa para solucionar esse problema, podemos sugerir a evolução da extensão *contains*, que não será abordado neste trabalho, mas será discutida como um trabalho futuro na Seção 6.2.

5.2 Avaliação do Trabalho

A Tabela 5.1 mostra a diferença entre as duas implementações para o designador *contains*, nas linguagens *eaj* e XAJ, em termos de linhas de código e módulos modificados. Note que o maior ganho do mecanismo proposto, quando comparado com *eaj*, está na quantidade de módulos modificados diretamente pelo usuário para que a extensão *contains* se torne funcional. A extensão desenvolvida em *eaj* teve a modificação de quatro módulos enquanto a desenvolvida neste trabalho, de apenas um. Com melhorias futuras, especialmente com a inclusão de programação generativa para a declaração @onWeaving, também podemos evoluir bastante em questão de linhas de código, fator que não foi nossa maior contribuição neste trabalho.

Tabela 5.1. Diferenças entre as abordagens para *contains*

Linguagem	Linhas de Código	Módulos Modificados
XAJ	$\approx$160	1
eaj	$\approx$190	4

Capítulo 6

Conclusões e Trabalhos Futuros

Este capítulo se destina a relatar os resultados obtidos com o presente trabalho e os benefícios alcançados, além de mostrar sugestões para trabalhos que poderão ser realizados, futuramente, com a linguagem XAJ.

6.1 Conclusões

Este trabalho apresentou a implementação de um mecanismo para extensibilidade de AspectJ em tempo de costura que foi solução para a criação de extensões para novos designadores de *pointcut* estáticos definidos pelo usuário. Embora haja outras abordagens que permitam a definição de novos *pointcuts* pelo usuário, este trabalho se concentrou em oferecer funcionalidades dentro de um compilador completo para AspectJ, o compilador modificado *AspectBench* para XAJ. Muitas abordagens, tais como Josh, geram novas linguagens que não são compatíveis com os recursos nativos de AspectJ. Esse não é o caso da linguagem XAJ, nem o propósito do nosso trabalho. Mantemos a compatibilidade dos recursos nativos de AspectJ em XAJ. O fato de se basear na linguagem orientada a aspectos mais popular pode tornar as características propostas disponíveis para um maior número de usuários potenciais.

Levando em conta as restrições impostas, consideramos que o trabalho atingiu parcialmente os objetivos planejados: tivemos um bom ganho em modularidade na implementação de uma extensão, mas não o suficiente em termos de linhas de código porém, ainda nos limitamos a designadores que usam apenas informações estáticas do programa base. Problemas encontrados poderão ser objeto de trabalhos futuros e são discutidos a seguir.

6.2 Problemas Encontrados e Trabalhos Futuros

Em Di Iorio et al. [2009] foi proposto um método a ser chamado em tempo de costura para cada ponto de junção atingido e visitado pelo costurador. O método *onWeaving* usaria programação generativa para geração de código e retornaria uma AST modificada para tal construção atingida, propondo uma separação clara de processamento em tempo de execução e em tempo de costura.

extensão mais geral, com uma abordagem não-restritiva. Métodos auxiliares muitas vezes são necessários para a construção de uma extensão. A própria extensão *contains* de *eaj* utiliza-se de um método auxiliar *doShadows*, que não poderia ser escrito na proposta do método *onWeaving*. Por esse motivo, tal proposta se tornou obsoleta, sendo substituída pela criação de uma nova anotação semelhante as já existentes na linguagem XAJ, a anotação @onWeaving. A nova anotação permite a definição desses métodos auxiliares, o que não era possível na proposta feita inicialmente. A ausência deste recurso tornaria o código muito complicado e ilegível.

A extensão *IsType*, mencionada na Seção 2.6, caracteriza um designador de *pointcut* dinâmico, que pode gerar código residual, precisando de validações com informações obtidas somente durante o tempo de execução. Para extensões desse tipo devem ser construídas classes que atuam como geradoras de códigos residuais, e a *syntaxclass* necessitaria de construções mais elaboradas ainda não implementadas.

O designador de *pointcut contains* não necessita dessa construção por ser um designador estático. A *syntaxclass* de XAJ ainda não está preparada para a criação e inserção de resíduos e otimizações no código costurado, para extensões nela descritas. Isto pode ser realizado em trabalhos futuros.

Para eliminar o problema discutido na Seção 5.1, referente a captura de contexto pelo designador *contains* de métodos mais internos, chamados por meio de outros, propomos uma extensão denominada *containsdeep*. Este designador olharia em profundidade na pilha de execução a sequência de chamadas para funções. Assim, conseguiríamos capturar o método *moveFigureAll*, ao invés dos específicos para cada *Figure*. Esse designador receberia, da mesma forma que o *contains* um *pointcut* como parâmetro e, opcionalmente, a profundidade de busca, pois pode haver casos em que o usuário não necessite capturar o método mais externo e sim, um outro mais interno. O designador *containsdeep*, diferentemente do *contains*, é aqui proposto a ser um designador dinâmico, que usaria informações obtidas somente em tempo de execução. Uma melhor análise poderia refutar a proposta inicial e levar a implementação para outro lado, de modo a construir o *containsdeep* como um designador estático.

O método de extensão do tempo de costura apresentado e a representação da AST

são bastante dependentes do compilador em questão. Como mostrado na Seção 3.2, as construções usam classes específicas do compilador *AspectBench*. Assim, também podemos citar como trabalho futuro o estudo de metodologias para especificar soluções que sejam menos dependentes do compilador utilizado. Quando escrevemos uma solução em XAJ temos que especificar relações de herança e usar classes pré-existentes. Estas classes existem tanto no *abc* quanto em outros compiladores de AspectJ e têm funções semelhantes. Embora não totalmente iguais, as classes poderiam ser usadas como coringas ao invés de serem especificadas diretamente nos arquivos escritos em XAJ podendo, dessa forma, reconhecer qual o compilador usado e usar suas classes pré-definidas.

Referências Bibliográficas

Aotani, T. & Masuhara, H. (2007). Scope: an aspectj compiler for supporting user-defined analysis-based pointcuts. Em *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pp. 161--172, New York, NY, USA. ACM.

AspectJ, E. (2012). The AspectJ Compiler. `http://www.eclipse.org/aspectj/doc/next/devguide/ajc-ref.html`. Acessado em 18/08/2012.

Avgustinov, P.; Christensen, A. S.; Hendren, L.; Kuzins, S.; Lhoták, J.; Lhoták, O.; de Moor, O.; Sereni, D.; Sittampalam, G. & Tibble, J. (2005). abc: an extensible aspectj compiler. Em *Proceedings of the 4th international conference on Aspect-oriented software development*, AOSD '05, pp. 87--98, New York, NY, USA. ACM.

Berk, E. (1997). *JLex: A lexical analyser generator for Java*. Departament of Computer Science, Princeton University.

Breuel, C. & Reverbel, F. (2007). Join point selectors. Em *Proceedings of the 5th workshop on Software engineering properties of languages and aspect technologies*, SPLAT '07, New York, NY, USA. ACM.

Brukman, M. & Myers, A. C. (2008). PPG: A parser generator for extensible grammars. http://www.cs.cornell.edu/Projects/polyglot/ppg.html.

Chiba, S. & Nakagawa, K. (2004). Josh: an open aspectj-like language. Em *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pp. 102--111, New York, NY, USA. ACM.

Chiba, S. & Nishizawa, M. (2003). An easy-to-use toolkit for efficient java bytecode translators. Em *In 2nd International coference on Generative Programming and Component Engineering (GPCE '03), volume 2830 of Springer Lecture Notes in Computer Science*, pp. 364--376. Springer-Verlag.

Clark., T. (2009). First class grammars for language oriented programming. Em *The 13th World Multi-Conference on Systemics, Cybernetics and Informatics: WMSCI 2009*.

Clark, T.; Sammut, P. & Willans, J. (2008). Beyond Annotations: A Proposal for Extensible Java (XJ). (http://www.ceteva.com/docs/XJ.pdf).

Di Iorio, V. O.; Reis, L. V.; Bigonha, R. S. & Valente, M. T. O. (2011). Weave time macros. Em *Proceedings of the sixth annual workshop on Domain-specific aspect languages*, DSAL '11, pp. 1--3, New York, NY, USA. ACM.

Di Iorio, V. O.; Reis, L. V. d. S.; Bigonha, R. d. S. & Bigonha, M. A. d. S. (2009). A proposal for extensible AspectJ. Em *DSAL '09: Proceedings of the 4th workshop on Domain-specific aspect languages*, pp. 21--24, New York, NY, USA. ACM.

Di Iorio, V. O.; Reis, L. V. d. S.; Trevenzoli, C. & Amorim, L. E. d. S. (2010). Implementation of user-defined pointcuts in the XAJ language. Em *Proceedings of the IV Latin American Workshop on Aspect-Oriented Software Development*, volume 9, pp. 43--48.

Eclipse (2012). Appendix A. AspectJ Quick Reference. `http://www.eclipse.org/aspectj/doc/released/progguide/quick.html`. Acessado em 21/08/2012.

Gamma, E.; Helm, R.; Johnson, R. & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, Bookman.

Hudson, S. (1999). *CUP User's Manual*. Georgia Institute of Technology.

Hudson, S. E.; Flannery, F.; Ananian, C. S.; Wang, D. & Appel, A. (1996). CUP LALR parser generator for java. Located at http://www.cs.princeton.edu/ appel/modern/java/-CUP/.

JBoss (2012). JBoss Inc., JBoss AOP Reference Documentation. (http://docs.jboss.org/jbossaop/docs/).

Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J. & Griswold, W. G. (2001). An overview of aspectj. Em *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pp. 327--353, London, UK. Springer-Verlag.

Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C. V.; Loingtier, J.-M. & Irwin, J. (1997). Aspect-oriented programming. Em *ECOOP*, pp. 220–242.

Miecznikowski, J. & Hendren, L. J. (2002). Decompiling java bytecode: Problems, traps and pitfalls. Em *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pp. 111--127, London, UK, UK. Springer-Verlag.

Nystrom, N.; Clarkson, M. R. & Myers, A. C. (2003). Polyglot: An extensible compiler framework for java. Em *In 12th International Conference on Compiler Construction*, pp. 138--152. Springer-Verlag.

POA (2012). Programação Orientada a Aspecto. `http://pt.wikipedia.org/wiki/Programa%C3%A7%C3%A3o_orientada_a_aspecto`. Acessado em 11/09/2012.

Reis, L. V. d. S.; Di Iorio, V. O.; Bigonha, R. d. S.; Bigonha, M. A. d. S. & Ladeira, R. d. C. (2009). XAJ: An extensible aspect-oriented language. Em *Proceedings of the III Latin American Workshop on Aspect-Oriented Software Development*, pp. 57--62. Federal University of Ceará.

Störzer, M. & Koppen, C. (2004). Pcdiff: Attacking the fragile pointcut problem, abstract. Em *European Interactive Workshop on Aspects in Software*, Berlin, Germany.

Tourwe, T.; Brichau, J. & Gybels, K. (2003). On the existence of the aosd-evolution paradox. Em *SPLAT: Software engineering Properties of Languages for Aspect Technologies*.

Vallée-Rai, R.; Co, P.; Gagnon, E.; Hendren, L.; Lam, P. & Sundaresan, V. (1999). Soot - a java bytecode optimization framework. Em *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99, pp. 13--. IBM Press.

Varejão, F. (2004). *Linguagens de programação Java, C e C++ e outras: conceitos e técnicas*, pp. 6--12. Campus, Rio de Janeiro.

Apêndice A

Gramática da Linguagem XAJ

A gramática da linguagem XAJ é definida a partir da gramática da linguagem AspectJ que, por fim, é definida a partir da gramática de Java. A gramática da linguagem na forma em que foi utilizada para gerar a tabela de *parser* e o mecanismo automático definido por @onWeaving é apresentada abaixo. As definições das regras estão no formato esperado pelo pré-processador para o gerador de analisadores sintáticos LALR CUP Hudson et al. [1996], o pré-processador PPG Brukman & Myers [2008] do Polyglot, que permite estender uma gramática escrita tanto em CUP quanto em PPG. A principal regra usada na definição da gramática XAJ é

extends S ::= <productions>;

que adiciona as produções especificadas por <*productions*> às geradas pelo não terminal *S*. Assim, conseguimos estender *type_declaration* e *class_member_declaration*, que são não terminais da gramática de Java, e definir, a partir deles, as novas produções requeridas por XAJ. As gramáticas das linguagens AspectJ e Java que são usadas na definição da gramática de XAJ podem ser encontradas nos apêndices B e C, respectivamente.

Gramática da Linguagem XAJ

```
// ------- Productions ----------------------------

start with goal;

/* add the possiblility of declaring a syntax class to type_declaration */
extend type_declaration ::=   // class | interface | aspect | syntaxclass
     syntaxclass_declaration;

/* add the possiblility od declaring a syntaclass as class member */
extend class_member_declaration ::=
```

```
11       syntaxclass_declaration;
12
13  syntaxclass_declaration ::=
14       modifiers_opt SYNTAXCLASS IDENTIFIER super_opt interfaces_opt syntaxclass_body;
15
16  syntaxclass_body ::=
17       LBRACE numberofpasses_opt syntax_declaration on_weaving_opt
18                  class_body_declarations_opt RBRACE;
19
20  numberofpasses_opt ::=
21       NUMBEROFPASSES EQ INTEGER_LITERAL SEMICOLON
22      |
23       ;
24
25  on_weaving_opt ::=
26           ONWEAVING IDENTIFIER on_weaving_declaration_body
27      |
28       ;
29
30  on_weaving_declaration_body ::=
31       LBRACE class_body_declarations_opt RBRACE;
32
33  syntax_declaration ::=
34       GRAMMAR using_opt syntax_declaration_body
35      |
36       GRAMMAR EXTENDS IDENTIFIER using_opt syntax_declaration_body
37      |
38       GRAMMAR OVERRIDES IDENTIFIER SEMICOLON;
39
40  using_opt ::=
41       USING list_identifier
42      |
43       ;
44
45  list_identifier ::=
46       list_identifier COMMA IDENTIFIER
47      |
48       IDENTIFIER;
49
50  syntax_declaration_body ::=
51       LBRACE grammar_productions RBRACE;
52
53  grammar_productions ::=
54       grammar_productions IDENTIFIER ARROW grammar_expression SEMICOLON
```

```
55      |
56       IDENTIFIER ARROW grammar_expression SEMICOLON;
57
58  grammar_expression ::=
59       grammar_expression OR production_element
60      |
61       production_element;
62
63  production_element ::=
64           grammar_term semantic_action_opt
65      |
66       /* LAMBDA followed by optional semantic action */
67       semantic_action_opt;
68
69  grammar_term ::=
70       grammar_term grammar_factor
71      |
72       grammar_factor;
73
74  grammar_factor ::=
75       IDENTIFIER:
76      |
77       IDENTIFIER EQ IDENTIFIER:
78      |
79       STRING_LITERAL
80      |
81       LPAREN grammar_expr RPAREN
82      |
83       named_opt LPAREN grammar_expr RPAREN
84      |
85       LBRACK grammar_expr RBRACK
86      |
87       named_opt LBRACK grammar_expr RBRACK
88      |
89       LBRACE grammar_expr RBRACE
90      |
91       named_opt LBRACE grammar_expr RBRACE;
92
93  named_opt ::=
94       IDENTIFIER EQ;
95
96  grammar_expr ::=
97       grammar_expr OR grammar_term
98      |
```

```
99        grammar_term
100      |
101       /* LAMBDA as a grammar_term */
102       ;
103
104  semantic_action_opt ::=
105       LBRACE COLON block_statements_opt COLON RBRACE
106      |
107       ;
108
109  extend import_declaration ::=
110       import_syntax
111      |
112       import_symbol;
113
114  import_syntax ::=
115       IMPORTSYNTAX qualified_name SEMICOLON
116      |
117       IMPORTSYNTAX name DOT MULT SEMICOLON;
118
119  import_symbol ::=
120       IMPORTSYMBOL qualified_name SEMICOLON
121      |
122       IMPORTSYMBOL name DOT MULT SEMICOLON;
```

Apêndice B

Gramática da Linguagem AspectJ

O compilador de XAJ é baseado no *abc*. Logo, a gramática da linguagem AspectJ usada neste trabalho é a disponibilizada pelos desenvolvedores desse compilador. Essa gramática é a usada para gerar o analisador sintático. A gramática da linguagem AspectJ estende a gramática da linguagem Java e é apresentada abaixo.

Gramática da Linguagem AspectJ

```
1  /* abc - The AspectBench Compiler
2   * Copyright (C) 2004 Laurie Hendren
3   * Copyright (C) 2004 Oege de Moor
4   * Copyright (C) 2004 Aske Simon Christensen
5   *
6   * This compiler is free software; you can redistribute it and/or
7   * modify it under the terms of the GNU Lesser General Public
8   * License as published by the Free Software Foundation; either
9   * version 2.1 of the License, or (at your option) any later version.
10  *
11  * This compiler is distributed in the hope that it will be useful,
12  * but WITHOUT ANY WARRANTY; without even the implied warranty of
13  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
14  * Lesser General Public License for more details.
15  *
16  * You should have received a copy of the GNU Lesser General Public
17  * License along with this compiler, in the file LESSER-GPL;
18  * if not, write to the Free Software Foundation, Inc.,
19  * 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
20  */
21
22 /**
23 * @author Laurie Hendren
24 * @author Oege de Moor
```

```
25  * @author Aske Simon Christensen
26  */
27
28  start with goal;
29
30  /* ------------------------------------------------------------ */
31  /*             EXTENSIONS TO BASE JAVA RULES                     */
32  /* ------------------------------------------------------------ */
33
34  /* add the possiblility of declaring an apspect to type_declaration */
35  extend type_declaration ::=   // class | interface | aspect
36       aspect_declaration;
37
38  /* add the possibility of a call to proceed as a method_invocation */
39  extend method_invocation ::=
40       PROCEED LPAREN argument_list_opt RPAREN;
41
42  /* add the possiblity of a simple_name for this_ aspectj vars */
43  /*
44  extend simple_name ::=
45       THISJOINPOINT
46    |
47       THISJOINPOINTSTATICPART
48    |
49       THISENCLOSINGJOINPOINTSTATICPART;
50    */
51
52
53  extend class_member_declaration ::=
54      aspect_declaration
55  |
56      pointcut_declaration;
57
58  extend interface_member_declaration ::=
59      aspect_declaration
60  |
61     pointcut_declaration;
62
63  /* ------------------------------------------------------------ */
64  /*                      ASPECTJ RULES                            */
65  /* ------------------------------------------------------------ */
66
67  /* must explicitly give two alternatives here,  if you make PRIVILEGED
68     another rule which can go to epsilon,  then there is a shift reduce
```

```
69      conflict with modifiers_opt, which can also go to epsilon. */
70
71  aspect_declaration ::=
72        modifiers_opt PRIVILEGED modifiers_opt ASPECT identifier
73                          super_opt interfaces_opt  perclause_opt
74                          aspect_body
75      |
76        modifiers_opt ASPECT identifier
77                          super_opt interfaces_opt  perclause_opt
78                          aspect_body;
79
80  perclause_opt ::=
81       perclause
82     |
83     /* epsilon */
84        ;
85
86  perclause ::=
87       PERTARGET LPAREN pointcut_expr RPAREN
88     |
89       PERTHIS LPAREN pointcut_expr RPAREN
90     |
91       PERCFLOW LPAREN pointcut_expr RPAREN
92     |
93       PERCFLOWBELOW LPAREN pointcut_expr RPAREN
94     |
95       ISSINGLETON
96     |
97       ISSINGLETON LPAREN RPAREN;
98
99  aspect_body ::=
100      LBRACE  RBRACE
101    |
102      LBRACE aspect_body_declarations RBRACE;
103
104 aspect_body_declarations ::=
105      aspect_body_declaration
106    |
107      aspect_body_declarations aspect_body_declaration;
108
109 aspect_body_declaration ::=
110      class_body_declaration
111    |
112      declare_declaration
```

```
113 /*|
114       pointcut_declaration*/
115    |
116       advice_declaration
117    |
118       intertype_member_declaration;
119
120 declare_declaration ::=
121       DECLARE PC_PARENTS COLON classname_pattern_expr
122              EXTENDS interface_type_list SEMICOLON
123    |
124       DECLARE PC_PARENTS COLON classname_pattern_expr
125              IMPLEMENTS interface_type_list SEMICOLON
126    |
127       DECLARE PC_WARNING COLON pointcut_expr COLON STRING_LITERAL SEMICOLON
128    |
129       DECLARE PC_ERROR COLON pointcut_expr COLON STRING_LITERAL SEMICOLON
130    |
131       DECLARE PC_SOFT COLON type COLON pointcut_expr SEMICOLON
132    |
133       DECLARE PC_PRECEDENCE COLON classname_pattern_expr_list SEMICOLON;
134
135 pointcut_declaration ::=
136       modifiers_opt POINTCUT identifier
137               LPAREN formal_parameter_list_opt RPAREN SEMICOLON
138    |
139       modifiers_opt POINTCUT identifier
140               LPAREN formal_parameter_list_opt RPAREN
141               COLON pointcut_expr SEMICOLON;
142
143 advice_declaration ::=
144        modifiers_opt advice_spec throws_opt COLON pointcut_expr
145       /* only valid modifier is strictfp */
146          method_body;
147
148 advice_spec ::=
149       BEFORE LPAREN formal_parameter_list_opt RPAREN
150    |
151       AFTER LPAREN formal_parameter_list_opt RPAREN
152    |
153       AFTER LPAREN formal_parameter_list_opt RPAREN PC_RETURNING
154    |
155       AFTER LPAREN formal_parameter_list_opt
156             RPAREN PC_RETURNING LPAREN RPAREN
```

```
157     |
158        AFTER LPAREN formal_parameter_list_opt RPAREN PC_RETURNING
159              LPAREN formal_parameter RPAREN
160     |
161        AFTER LPAREN formal_parameter_list_opt RPAREN PC_THROWING
162     |
163        AFTER LPAREN formal_parameter_list_opt RPAREN PC_THROWING LPAREN RPAREN
164     |
165        AFTER LPAREN formal_parameter_list_opt RPAREN PC_THROWING
166              LPAREN formal_parameter RPAREN
167     |
168        type AROUND LPAREN formal_parameter_list_opt RPAREN
169     |
170        VOID AROUND LPAREN formal_parameter_list_opt RPAREN:y
171          {: Grm.parserTrace("VOID around (formals)");
172             TypeNode voidn =  parser.nf.CanonicalTypeNode(parser.pos(a),
173                                                   parser.ts.Void());
174             Around around = parser.nf.Around(parser.pos(a,y),
175                                         voidn,
176                                         b);
177             RESULT = around;
178          :}
179     ;
180
181   intertype_member_declaration ::=
182        modifiers_opt VOID name DOT identifier
183            LPAREN formal_parameter_list_opt RPAREN throws_opt method_body
184     |
185        modifiers_opt type name DOT identifier
186            LPAREN formal_parameter_list_opt RPAREN throws_opt method_body
187     |
188        modifiers_opt name DOT NEW LPAREN formal_parameter_list_opt
189            RPAREN throws_opt constructor_body
190     |
191        modifiers_opt type name DOT identifier SEMICOLON
192     |
193        modifiers_opt type name DOT identifier
194           EQ variable_initializer SEMICOLON;
195
196   /* ------------------------ POINTCUT EXPRESSIONS ---------------------- */
197
198   pointcut_expr ::=
199       and_pointcut_expr
200    |
```

```
201      pointcut_expr PC_OROR and_pointcut_expr;
202
203  and_pointcut_expr ::=
204      unary_pointcut_expr
205    |
206      and_pointcut_expr PC_ANDAND unary_pointcut_expr;
207
208  unary_pointcut_expr ::=
209        basic_pointcut_expr
210      |
211        PC_NOT unary_pointcut_expr;
212
213  basic_pointcut_expr ::=
214       LPAREN pointcut_expr RPAREN
215      |
216       PC_CALL LPAREN method_constructor_pattern RPAREN
217      |
218       PC_EXECUTION LPAREN method_constructor_pattern RPAREN
219      |
220       PC_INITIALIZATION LPAREN constructor_pattern RPAREN
221      |
222       PC_PREINITIALIZATION LPAREN constructor_pattern RPAREN
223      |
224       PC_STATICINITIALIZATION LPAREN classname_pattern_expr RPAREN
225      |
226       PC_GET LPAREN field_pattern RPAREN
227      |
228       PC_SET LPAREN field_pattern RPAREN
229      |
230       PC_HANDLER LPAREN classname_pattern_expr RPAREN
231      |
232       PC_ADVICEEXECUTION LPAREN RPAREN
233      |
234       PC_WITHIN LPAREN classname_pattern_expr RPAREN
235      |
236       PC_WITHINCODE LPAREN method_constructor_pattern RPAREN
237      |
238       PC_CFLOW LPAREN pointcut_expr RPAREN
239      |
240       PC_CFLOWBELOW LPAREN pointcut_expr RPAREN
241      |
242       PC_IF LPAREN expression RPAREN
243      |
244       PC_THIS LPAREN type_id_star RPAREN
```

```
245     |
246       PC_TARGET LPAREN type_id_star RPAREN
247     |
248       PC_ARGS LPAREN type_id_star_list_opt RPAREN
249     |
250       name LPAREN type_id_star_list_opt RPAREN;
251
252 /* ------------------- NAME PATTERNS ---------------------------------- */
253
254 name_pattern ::=
255        simple_name_pattern
256     |
257        name_pattern DOT simple_name_pattern
258     |
259        name_pattern PC_DOTDOT simple_name_pattern;
260
261 simple_name_pattern ::=
262       PC_MULT
263     |
264       IDENTIFIERPATTERN
265     |
266       identifier
267     |
268       aspectj_reserved_identifier;
269
270 aspectj_reserved_identifier ::=
271       ASPECT
272    |
273       PRIVILEGED
274    |
275       PC_ADVICEEXECUTION
276    |
277       PC_ARGS
278    |
279       PC_CALL
280    |
281       PC_CFLOW
282    |
283       PC_CFLOWBELOW
284    |
285       PC_ERROR
286    |
287       PC_EXECUTION
288    |
```

```
289         PC_GET
290     |
291         PC_HANDLER
292     |
293         PC_INITIALIZATION
294     |
295         PC_PARENTS
296     |
297         PC_PRECEDENCE
298     |
299         PC_PREINITIALIZATION
300     |
301         PC_RETURNING
302     |
303         PC_SET
304     |
305         PC_SOFT
306     |   PC_STATICINITIALIZATION
307     |
308         PC_TARGET
309     |
310         PC_THROWING
311     |
312         PC_WARNING
313     |
314         PC_WITHINCODE;
315
316 classtype_dot_id ::=
317         simple_name_pattern
318     |
319         name_pattern DOT simple_name_pattern
320     |
321         name_pattern PC_PLUS DOT simple_name_pattern
322     |
323         name_pattern PC_DOTDOT simple_name_pattern
324     |
325         LPAREN type_pattern_expr RPAREN DOT simple_name_pattern;
326
327 classtype_dot_new ::=
328         NEW
329     |
330         name_pattern DOT NEW
331     |
332         name_pattern PC_PLUS DOT NEW
```

```
333    |
334      name_pattern PC_DOTDOT NEW
335    |
336      LPAREN type_pattern_expr RPAREN DOT NEW;
337
338 /* ------------------- TYPE PATTERNS ---------------------------------- */
339
340
341 type_pattern_expr ::=
342       or_type_pattern_expr
343     |
344       type_pattern_expr PC_ANDAND or_type_pattern_expr;
345
346 or_type_pattern_expr ::=
347       unary_type_pattern_expr
348     |
349       or_type_pattern_expr PC_OROR unary_type_pattern_expr;
350
351 unary_type_pattern_expr ::=
352       basic_type_pattern
353     |
354       PC_NOT unary_type_pattern_expr;
355
356 /* check that VOID is not in patterns for formals, ok for
357        patterns for return types */
358 basic_type_pattern ::=
359      VOID
360    |
361      base_type_pattern
362    |
363      base_type_pattern dims
364    |
365      LPAREN type_pattern_expr RPAREN;
366
367 base_type_pattern ::=
368      primitive_type
369    |
370      name_pattern
371    |
372      name_pattern PC_PLUS;
373
374 /* ------------------ CLASSNAME PATTERNS ------------------------------- */
375
376 classname_pattern_expr_list ::=
```

```
377     classname_pattern_expr
378   |
379     classname_pattern_expr_list COMMA classname_pattern_expr;
380
381 classname_pattern_expr ::=
382      and_classname_pattern_expr
383    |
384      classname_pattern_expr PC_OROR and_classname_pattern_expr;
385
386 and_classname_pattern_expr ::=
387      unary_classname_pattern_expr
388    |
389      and_classname_pattern_expr PC_ANDAND unary_classname_pattern_expr;
390
391 unary_classname_pattern_expr ::=
392      basic_classname_pattern
393    |
394      PC_NOT unary_classname_pattern_expr;
395
396 basic_classname_pattern ::=
397     name_pattern
398   |
399     name_pattern PC_PLUS
400   |
401     LPAREN classname_pattern_expr RPAREN;
402
403 classname_pattern_expr_nobang ::=
404      and_classname_pattern_expr_nobang
405    |
406      classname_pattern_expr_nobang PC_OROR and_classname_pattern_expr;
407
408 and_classname_pattern_expr_nobang ::=
409      basic_classname_pattern
410    |
411      and_classname_pattern_expr_nobang PC_ANDAND unary_classname_pattern_expr;
412
413 /* ---------------------- MODIFIER PATTERNS --------------------------- */
414
415 modifier_pattern_expr ::=
416     modifier
417   |
418     PC_NOT modifier
419   |
420     modifier_pattern_expr modifier
```

```
421   |
422       modifier_pattern_expr PC_NOT modifier;
423
424 /* -------------- METHOD, CONSTRUCTOR and FIELD PATTERNS ----------------- */
425
426 throws_pattern_list_opt ::=
427       THROWS throws_pattern_list
428   |
429       // epsilon
430        ;
431 throws_pattern_list ::=
432       throws_pattern
433   |
434       throws_pattern_list COMMA throws_pattern;
435
436 throws_pattern ::=
437       classname_pattern_expr_nobang
438   |
439       PC_NOT classname_pattern_expr;
440
441 method_constructor_pattern ::=
442       method_pattern
443   |
444       constructor_pattern;
445
446 method_pattern ::=
447       modifier_pattern_expr
448         type_pattern_expr
449         classtype_dot_id
450         LPAREN formal_pattern_list_opt RPAREN
451         throws_pattern_list_opt
452   |
453       type_pattern_expr classtype_dot_id
454         LPAREN formal_pattern_list_opt RPAREN
455         throws_pattern_list_opt;
456
457 constructor_pattern ::=
458       modifier_pattern_expr
459         classtype_dot_new
460         LPAREN formal_pattern_list_opt RPAREN
461         throws_pattern_list_opt
462   |
463         classtype_dot_new
464         LPAREN formal_pattern_list_opt RPAREN
```

```
465         throws_pattern_list_opt;
466
467 field_pattern ::=
468      modifier_pattern_expr type_pattern_expr classtype_dot_id
469   |
470      type_pattern_expr classtype_dot_id;
471
472 /* -------------------- FORMAL PARAMETER LIST PATTERNS --------------*/
473
474 formal_pattern_list_opt ::=
475       formal_pattern_list
476   |
477       // epsilon
478        ;
479 formal_pattern_list ::=
480       formal_pattern
481   |
482       formal_pattern_list COMMA formal_pattern;
483
484 formal_pattern ::=
485        PC_DOTDOT
486     |
487        DOT DOT
488     |
489        type_pattern_expr;
490
491 /* -------------------- POINTCUT PARAMETER LIST PATTERNS --------------*/
492
493 type_id_star_list_opt ::=
494       type_id_star_list
495     |
496       // epsilon
497        ;
498
499 type_id_star_list ::=
500       type_id_star
501     |
502       type_id_star_list COMMA type_id_star;
503
504 // there should be three alternatives here: star, type, and identifier
505 // disambiguation between type and identifier happens in the type-checker
506 type_id_star ::=
507        PC_MULT
508     |
```

```
509        PC_DOTDOT
510      |
511        type
512      |
513        type PC_PLUS;
```

Apêndice C

Gramática da Linguagem Java

A seguir é apresentada a gramática da linguagem Java, fornecida pelos desenvolvedores do Polyglot, usada como referência neste trabalho para gerar o analisador sintático. A versão utilizada é a 1.2 estendida da 1.4 para suportar asserções.

Gramática da Linguagem Java

```
1  // This parser is based on:
2  /* Java 1.2 parser for CUP.
3   * Copyright (C) 1998 C. Scott Ananian <cananian@alumni.princeton.edu>
4   * This program is released under the terms of the GPL; see the file
5   * COPYING for more details.  There is NO WARRANTY on this code.
6   *
7   * As a special exception, C. Scott Ananian additionally permits the
8   * distribution of this modified version of the parser and its derivatives
9   * under the terms of the LGPL.
10  *
11  * extended to support Java 1.4 (assert keyword).
12  */
13
14 start with goal;
15
16 // 19.2) The Syntactic Grammar
17 goal ::=
18                     // SourceFile
19         compilation_unit;
20
21 // 19.3) Lexical Structure.
22 literal ::=
23                     // Lit
24         INTEGER_LITERAL
25     |
```

```
26          LONG_LITERAL
27      |
28          DOUBLE_LITERAL
29      |
30          FLOAT_LITERAL
31      |
32          BOOLEAN_LITERAL
33      |
34          CHARACTER_LITERAL
35      |
36          STRING_LITERAL
37      |
38          NULL_LITERAL;
39  boundary_literal ::=
40                      // Lit
41          INTEGER_LITERAL_BD
42      |
43          LONG_LITERAL_BD;
44
45
46  // 19.4) Types, Values, and Variables
47  type ::=
48                      // TypeNode
49          primitive_type
50      |
51         reference_type;
52  primitive_type ::=
53                      // TypeNode
54          numeric_type
55      |
56          BOOLEAN;
57  numeric_type::=
58                      // TypeNode
59          integral_type
60      |
61         floating_point_type;
62  integral_type ::=
63                      // TypeNode
64          BYTE
65      |
66          CHAR
67      |
68          SHORT
69      |
```

```
70          INT
71      |
72          LONG;
73  floating_point_type ::=
74                      // TypeNode
75          FLOAT
76      |
77          DOUBLE;
78  reference_type ::=
79                      // TypeNode
80          class_or_interface_type
81      |
82          array_type;
83  class_or_interface_type ::=
84                      // TypeNode
85          name;
86  class_type ::=
87                      // TypeNode
88          class_or_interface_type;
89  interface_type ::=
90                      // TypeNode
91          class_or_interface_type;
92  array_type ::=
93                      // TypeNode
94          primitive_type dims
95      |
96          name dims;
97  // 19.5) Names
98
99  // Add by Leonardo
100
101 identifier ::= IDENTIFIER;
102
103 name    ::=
104                     // Name
105         simple_name
106     |
107         qualified_name;
108 simple_name ::=
109                     // Name
110         identifier;
111 qualified_name ::=
112                     // Name
113         name DOT identifier;
```

```
114 // 19.6) Packages
115 compilation_unit ::=
116                     // SourceFile
117         package_declaration_opt
118         import_declarations_opt
119         type_declarations_opt
120     |
121         error
122         type_declarations_opt;
123 package_declaration_opt ::=
124                     // PackageNode
125         package_declaration
126     |
127          ;
128 import_declarations_opt ::=
129                     // List of Import
130         import_declarations
131     |
132          ;
133 type_declarations_opt   ::=
134                     // List of TopLevelDecl
135         type_declarations
136     |
137          ;
138 import_declarations ::=
139                     // List of Import
140         import_declaration
141     |
142         import_declarations import_declaration;
143 type_declarations ::=
144                     // List of TopLevelDecl
145         type_declaration
146     |
147         type_declarations type_declaration;
148 package_declaration ::=
149                     // PackageNode
150         PACKAGE name SEMICOLON;
151 import_declaration ::=
152                     // Import
153         single_type_import_declaration
154     |
155         type_import_on_demand_declaration;
156 single_type_import_declaration ::=
157                     // Import
```

```
158         IMPORT qualified_name SEMICOLON;
159 type_import_on_demand_declaration ::=
160                     // Import
161         IMPORT name DOT MULT SEMICOLON;
162 type_declaration ::=
163                     // ClassDecl
164         class_declaration
165     |
166         interface_declaration
167     |
168         SEMICOLON;
169
170 // 19.7) Productions used only in the LALR(1) grammar
171 modifiers_opt::=
172                     // Flags
173         modifiers
174     |
175          ;
176 modifiers ::=
177                     // Flags
178         modifier
179     |
180         modifiers modifier;
181 modifier ::=
182                     // Flags
183         PUBLIC
184     |
185         PROTECTED
186     |
187         PRIVATE
188     |
189         STATIC
190     |
191         ABSTRACT
192     |
193         FINAL
194     |
195         NATIVE
196     |
197         SYNCHRONIZED
198     |
199         TRANSIENT
200     |
201         VOLATILE
```

```
202     |
203         STRICTFP;
204 // 19.8) Classes
205
206 // 19.8.1) Class Declarations
207 class_declaration ::=
208                     // ClassDecl
209         modifiers_opt CLASS identifier;
210 super ::=
211                     // TypeNode
212         EXTENDS class_type;
213 super_opt ::=
214                    // TypeNode
215         super
216     |
217          ;
218 interfaces ::=
219                     // List of TypeNode
220         IMPLEMENTS interface_type_list;
221 interfaces_opt::=
222                     // List of TypeNode
223         interfaces
224     |
225          ;
226 interface_type_list ::=
227                     // List of TypeNode
228         interface_type
229     |
230         interface_type_list COMMA interface_type;
231 class_body ::=
232                     // ClassBody
233         LBRACE class_body_declarations_opt RBRACE;
234 class_body_declarations_opt ::=
235                     // List of ClassMember
236         class_body_declarations
237     |
238          ;
239 class_body_declarations ::=
240                     // List of ClassMember
241         class_body_declaration
242     |
243         class_body_declarations class_body_declaration;
244 class_body_declaration ::=
245                     // List of ClassMember
```

```
246         class_member_declaration
247     |
248         static_initializer
249     |
250         constructor_declaration
251     |
252         block
253     |
254         SEMICOLON
255     |
256         error SEMICOLON
257     |
258         error LBRACE;
259 class_member_declaration ::=
260                     // List of ClassMember
261         field_declaration
262     |
263         method_declaration
264         /* repeat the prod for 'class_declaration' here: */
265     |
266         modifiers_opt CLASS identifier
267     |
268         interface_declaration;
269
270 // 19.8.2) Field Declarations
271 field_declaration ::=
272                     // List of ClassMember
273         modifiers_opt type variable_declarators SEMICOLON;
274 variable_declarators ::=
275                     // List of VarDeclarator
276         variable_declarator
277     |
278         variable_declarators COMMA variable_declarator;
279 variable_declarator ::=
280                 // VarDeclarator
281         variable_declarator_id
282     |
283         variable_declarator_id EQ variable_initializer;
284 variable_declarator_id ::=
285                 // VarDeclarator
286         identifier
287     |
288         variable_declarator_id LBRACK RBRACK;
289 variable_initializer ::=
```

```
290                     // Expr
291         expression
292     |
293         array_initializer;
294
295 // 19.8.3) Method Declarations
296 method_declaration ::=
297                     // MethodDecl
298         method_header method_body;
299 method_header ::=
300                     // MethodDecl
301         modifiers_opt type identifier LPAREN
302                 formal_parameter_list_opt RPAREN dims_opt throws_opt
303     |
304         modifiers_opt VOID identifier LPAREN;
305 formal_parameter_list_opt ::=
306                     // List of Formal
307         formal_parameter_list
308     |
309          ;
310 formal_parameter_list ::=
311                     // List of Formal
312         formal_parameter
313     |
314         formal_parameter_list COMMA formal_parameter;
315 formal_parameter ::=
316                     // Formal
317         type variable_declarator_id
318     |
319         FINAL type variable_declarator_id;
320 throws_opt ::=
321                     // List of TypeNode
322         throws
323     |
324          ;
325 throws ::=
326                     // List of TypeNode
327         THROWS class_type_list;
328 class_type_list ::=
329                     // List of TypeNode
330         class_type
331     |
332         class_type_list COMMA class_type;
333 method_body ::=
```

```
334                     // Block
335         block
336     |
337         SEMICOLON;
338
339 // 19.8.4) Static Initializers
340 static_initializer ::=
341                     // Block
342         STATIC block;
343
344 // 19.8.5) Constructor Declarations
345 constructor_declaration ::=
346                     // ConstructorDecl
347         modifiers_opt simple_name LPAREN formal_parameter_list_opt RPAREN
348             throws_opt constructor_body;
349 constructor_body ::=
350                     // Block
351         LBRACE explicit_constructor_invocation block_statements RBRACE
352     |
353         LBRACE explicit_constructor_invocation RBRACE
354     |
355         LBRACE block_statements RBRACE
356     |
357         LBRACE RBRACE;
358 explicit_constructor_invocation ::=
359                     // ConstructorCall
360         THIS LPAREN argument_list_opt RPAREN SEMICOLON
361     |
362         SUPER LPAREN argument_list_opt RPAREN SEMICOLON
363     |
364         primary DOT THIS LPAREN argument_list_opt RPAREN SEMICOLON
365     |
366         primary DOT SUPER LPAREN argument_list_opt RPAREN SEMICOLON;
367
368 // 19.9) Interfaces
369
370 // 19.9.1) Interface Declarations
371 interface_declaration ::=
372                     // ClassDecl
373         modifiers_opt INTERFACE identifier;
374 extends_interfaces_opt ::=
375                     // List of TypeNode
376         extends_interfaces
377     |
```

```
378        ;
379 extends_interfaces ::=
380                     // List of TypeNode
381         EXTENDS interface_type
382     |
383         extends_interfaces COMMA interface_type;
384 interface_body ::=
385                     // ClassBody
386         LBRACE interface_member_declarations_opt RBRACE;
387 interface_member_declarations_opt ::=
388                     // List of ClassMember
389         interface_member_declarations
390     |
391        ;
392 interface_member_declarations ::=
393                     // List of ClassMember
394         interface_member_declaration
395     |
396         interface_member_declarations interface_member_declaration;
397 interface_member_declaration ::=
398                     // List of ClassMember
399         constant_declaration
400     |
401         abstract_method_declaration
402     |
403         class_declaration
404     |
405         interface_declaration
406     |
407         SEMICOLON;
408 constant_declaration ::=
409                     // List of ClassMember
410         field_declaration;
411 abstract_method_declaration ::=
412                     // MethodDecl
413         method_header SEMICOLON;
414
415 // 19.10) Arrays
416 array_initializer ::=
417                     // ArrayInit
418         LBRACE:n variable_initializers COMMA RBRACE
419     |
420         LBRACE variable_initializers RBRACE
421     |
```

```
422         LBRACE COMMA RBRACE
423     |
424         LBRACE RBRACE;
425 variable_initializers ::=
426                     // List of Expr
427         variable_initializer
428     |
429         variable_initializers COMMA variable_initializer;
430
431 // 19.11) Blocks and Statements
432 block ::=
433                     // Block
434         LBRACE block_statements_opt RBRACE
435     |
436         error RBRACE;
437 block_statements_opt ::=
438                     // List of Stmt
439         block_statements
440     |
441          ;
442 block_statements ::=
443                     // List of Stmt
444         block_statement
445     |
446         block_statements block_statement;
447 block_statement ::=
448                     // List of Stmt
449         local_variable_declaration_statement
450     |
451         statement
452     |
453         class_declaration;
454 local_variable_declaration_statement ::=
455                     // List of LocalDecl
456         local_variable_declaration SEMICOLON;
457 local_variable_declaration ::=
458                     // List of LocalDecl
459         type variable_declarators
460     |   FINAL type variable_declarators;
461 statement ::=
462                     // Stmt
463         statement_without_trailing_substatement
464     |
465         labeled_statement
```

```
466     |
467         if_then_statement
468     |
469         if_then_else_statement
470     |
471         while_statement
472     |
473         for_statement
474     |
475         error SEMICOLON;
476 statement_no_short_if ::=
477                 // Stmt
478         statement_without_trailing_substatement
479     |
480         labeled_statement_no_short_if
481     |
482         if_then_else_statement_no_short_if
483     |
484         while_statement_no_short_if
485     |
486         for_statement_no_short_if;
487 statement_without_trailing_substatement ::=
488                 // Stmt
489         block
490     |
491         empty_statement
492     |
493         expression_statement
494     |
495         switch_statement:
496     |
497         do_statement
498     |
499         break_statement
500     |
501         continue_statement
502     |
503         return_statement
504     |
505         synchronized_statement
506     |
507         throw_statement
508     |
509         try_statement
```

```
510     |
511         assert_statement;
512 empty_statement ::=
513                     // Empty
514         SEMICOLON;
515 labeled_statement ::=
516                     // Labeled
517         identifier COLON statement;
518 labeled_statement_no_short_if ::=
519                     // Labeled
520         identifier COLON statement_no_short_if;
521 expression_statement ::=
522                     // Stmt
523         statement_expression SEMICOLON;
524 statement_expression ::=
525                     // Expr
526         assignment
527     |
528         preincrement_expression
529     |
530         predecrement_expression
531     |
532         postincrement_expression
533     |
534         postdecrement_expression
535     |
536         method_invocation
537     |
538         class_instance_creation_expression;
539 if_then_statement ::=
540                     // If
541         IF LPAREN expression RPAREN statement;
542 if_then_else_statement ::=
543                     // If
544         IF LPAREN expression RPAREN statement_no_short_if;
545 if_then_else_statement_no_short_if ::=
546                     // If
547         IF LPAREN expression RPAREN statement_no_short_if
548             ELSE statement_no_short_if;
549 switch_statement ::=
550                     // Switch
551         SWITCH LPAREN expression RPAREN switch_block;
552 switch_block ::=
553                     // List of SwitchElement
```

```
554         LBRACE switch_block_statement_groups switch_labels RBRACE; :}
555     |
556         LBRACE switch_block_statement_groups RBRACE
557     |
558         LBRACE switch_labels RBRACE
559     |
560         LBRACE RBRACE;
561 switch_block_statement_groups ::=
562                     // List of SwitchElement
563         switch_block_statement_group
564     |
565         switch_block_statement_groups switch_block_statement_group;
566 switch_block_statement_group ::=
567                     // List of SwitchElement
568         switch_labels block_statements;
569 switch_labels ::=
570                     // List of Case
571         switch_label
572     |
573         switch_labels switch_label;
574 switch_label ::=
575                     // Case
576         CASE constant_expression COLON
577     |
578         DEFAULT COLON;
579
580 while_statement ::=
581                     // While
582         WHILE LPAREN expression RPAREN statement;
583 while_statement_no_short_if ::=
584                     // While
585         WHILE LPAREN expression RPAREN statement_no_short_if;
586 do_statement ::=
587                     // Do
588         DO statement WHILE LPAREN expression RPAREN SEMICOLON;
589 for_statement ::=
590                     // For
591         FOR LPAREN for_init_opt SEMICOLON expression_opt SEMICOLON
592             for_update_opt RPAREN statement;
593 for_statement_no_short_if ::=
594                     // For
595         FOR LPAREN for_init_opt SEMICOLON expression_opt SEMICOLON
596             for_update_opt RPAREN statement_no_short_if;
597 for_init_opt ::=
```

```
598                     // List of ForInit
599         for_init
600     |
601          ;
602 for_init ::=
603                     // List of ForInit
604         statement_expression_list
605     |
606         local_variable_declaration;
607 for_update_opt ::=
608                     // List of ForUpdate
609         for_update
610     |
611          ;
612 for_update ::=
613                     // List of ForUpdate
614         statement_expression_list;
615 statement_expression_list ::=
616                     // List of Stmt
617         statement_expression
618     |
619         statement_expression_list COMMA statement_expression;
620
621 identifier_opt ::=
622                     // Name
623         identifier
624     |
625          ;
626
627 break_statement ::=
628                     // Branch
629         BREAK identifier_opt SEMICOLON;
630
631 continue_statement ::=
632                     // Branch
633         CONTINUE identifier_opt SEMICOLON;
634 return_statement ::=
635                     // Return
636         RETURN expression_opt SEMICOLON;
637 throw_statement ::=
638                     // Throw
639         THROW expression SEMICOLON;
640 synchronized_statement ::=
641                     // Synchronized
```

```
642         SYNCHRONIZED LPAREN expression RPAREN block;
643 try_statement ::=
644                     // Try
645         TRY block catches
646     |
647         TRY block catches_opt finally;
648 catches_opt ::=
649                     // List of Catch
650         catches
651     |
652          ;
653 catches ::=
654                     // List of Catch
655         catch_clause
656     |
657         catches catch_clause;
658 catch_clause ::=
659                     // Catch
660         CATCH LPAREN formal_parameter RPAREN block;
661 finally ::=
662                     // Block
663         FINALLY block;
664
665 assert_statement ::=
666                     // Assert
667         ASSERT expression SEMICOLON
668     |
669         ASSERT expression COLON expression SEMICOLON;
670
671 // 19.12) Expressions
672 primary ::=
673                     // Expr
674         primary_no_new_array
675     |
676         array_creation_expression;
677 primary_no_new_array ::=
678                     // Expr
679         literal
680     |
681         THIS
682     |
683         LPAREN expression RPAREN
684     |
685         class_instance_creation_expression
```

```
686     |
687         field_access
688     |
689         method_invocation
690     |
691         array_access
692     |
693         primitive_type DOT CLASS
694     |
695         VOID DOT CLASS
696     |
697         array_type DOT CLASS
698     |
699         name DOT CLASS
700     |
701         name DOT THIS;
702 class_instance_creation_expression ::=
703                     // Expr
704         NEW class_type LPAREN argument_list_opt RPAREN
705     |
706         NEW class_type LPAREN argument_list_opt RPAREN class_body
707     |
708         primary DOT NEW simple_name LPAREN argument_list_opt RPAREN
709     |
710         primary DOT NEW simple_name LPAREN argument_list_opt RPAREN class_body
711     |
712         name DOT NEW simple_name LPAREN argument_list_opt RPAREN
713     |
714         name DOT NEW simple_name LPAREN argument_list_opt RPAREN class_body;
715 argument_list_opt ::=
716                     // List of Expr
717         argument_list
718     |
719          ;
720 argument_list ::=
721                     // List of Expr
722         expression
723     |
724         argument_list COMMA expression;
725 array_creation_expression ::=
726                     // NewArray
727         NEW primitive_type dim_exprs dims_opt
728     |
729         NEW class_or_interface_type dim_exprs dims_opt
```

```
730     |
731         NEW primitive_type dims array_initializer
732     |
733         NEW class_or_interface_type dims array_initializer;
734 dim_exprs ::=
735                     // List of Expr
736         dim_expr
737     |
738         dim_exprs dim_expr;
739 dim_expr ::=
740                     // Expr
741         LBRACK:x expression RBRACK;
742 dims_opt ::=
743                     // Integer
744         dims
745     |
746          ;
747 dims ::=
748                     // Integer
749         LBRACK RBRACK
750     |
751         dims LBRACK RBRACK;
752 field_access ::=
753                     // Field
754         primary DOT identifier
755     |
756         SUPER DOT identifier
757     |
758         name DOT SUPER DOT identifier;
759 method_invocation ::=
760                     // Call
761         name LPAREN argument_list_opt RPAREN
762     |
763         primary DOT identifier LPAREN argument_list_opt RPAREN
764     |
765         SUPER DOT identifier LPAREN argument_list_opt RPAREN
766     |
767         name DOT SUPER DOT identifier LPAREN argument_list_opt RPAREN;
768 array_access ::=
769                     // ArrayAccess
770         name LBRACK expression RBRACK
771     |
772         primary_no_new_array LBRACK expression RBRACK;
773 postfix_expression ::=
```

```
774                     // Expr
775         primary
776     |
777         name
778     |
779         postincrement_expression
780     |
781         postdecrement_expression;
782 postincrement_expression ::=
783                     // Unary
784         postfix_expression PLUSPLUS;
785 postdecrement_expression ::=
786                     // Unary
787         postfix_expression MINUSMINUS;
788 unary_expression ::=
789                     // Expr
790         preincrement_expression
791     |
792         predecrement_expression
793     |
794         PLUS unary_expression
795     |
796         MINUS unary_expression
797     |
798         MINUS boundary_literal
799     |
800         unary_expression_not_plus_minus;
801 preincrement_expression ::=
802                     // Unary
803         PLUSPLUS unary_expression;
804 predecrement_expression ::=
805                     // Unary
806         MINUSMINUS unary_expression;
807 unary_expression_not_plus_minus ::=
808                     // Expr
809         postfix_expression
810     |
811         COMP unary_expression
812     |
813         NOT unary_expression
814     |
815         cast_expression;
816 cast_expression ::=
817                     // Cast
```

```
818         LPAREN primitive_type dims_opt RPAREN unary_expression
819     |
820         LPAREN expression RPAREN unary_expression_not_plus_minus
821     |
822         LPAREN name dims RPAREN unary_expression_not_plus_minus;
823 multiplicative_expression ::=
824                     // Expr
825         unary_expression
826     |
827         multiplicative_expression MULT unary_expression
828     |
829         multiplicative_expression DIV unary_expression
830     |
831         multiplicative_expression MOD unary_expression;
832 additive_expression ::=
833                     // Expr
834         multiplicative_expression
835     |
836         additive_expression PLUS multiplicative_expression
837     |
838         additive_expression MINUS multiplicative_expression;
839 shift_expression ::=
840                     // Expr
841         additive_expression
842     |
843         shift_expression LSHIFT additive_expression
844     |
845         shift_expression RSHIFT additive_expression
846     |
847         shift_expression URSHIFT additive_expression;
848 relational_expression ::=
849                     // Expr
850         shift_expression
851     |
852         relational_expression LT shift_expression
853     |
854         relational_expression GT shift_expression
855     |
856         relational_expression LTEQ shift_expression
857     |
858         relational_expression GTEQ shift_expression
859     |
860         relational_expression INSTANCEOF reference_type;
861
```

```
862 equality_expression ::=
863                     // Expr
864         relational_expression
865     |
866         equality_expression EQEQ relational_expression
867     |
868         equality_expression NOTEQ relational_expression;
869 and_expression ::=
870                     // Expr
871         equality_expression
872     |
873         and_expression AND equality_expression;
874 exclusive_or_expression ::=
875                     // Expr
876         and_expression
877     |
878         exclusive_or_expression XOR and_expression;
879 inclusive_or_expression ::=
880                     // Expr
881         exclusive_or_expression
882     |
883         inclusive_or_expression OR exclusive_or_expression;
884 conditional_and_expression ::=
885                     // Expr
886         inclusive_or_expression
887     |
888         conditional_and_expression ANDAND inclusive_or_expression;
889 conditional_or_expression ::=
890                     // Expr
891         conditional_and_expression
892     |
893         conditional_or_expression OROR conditional_and_expression;
894 conditional_expression ::=
895                     // Expr
896         conditional_or_expression
897     |
898         conditional_or_expression QUESTION expression;
899 assignment_expression ::=
900                     // Expr
901         conditional_expression
902     |
903         assignment;
904 assignment ::=
905                     // Expr
```

```
906         left_hand_side assignment_operator assignment_expression;
907 left_hand_side ::=
908                     // Expr
909         name
910     |
911         field_access
912     |
913         array_access;
914 assignment_operator ::=
915                     // Assign.Operator
916         EQ
917     |
918         MULTEQ
919     |
920         DIVEQ
921     |
922         MODEQ
923     |
924         PLUSEQ
925     |
926         MINUSEQ
927     |
928         LSHIFTEQ
929     |
930         RSHIFTEQ
931     |
932         URSHIFTEQ
933     |
934         ANDEQ
935     |
936         XOREQ
937     |
938         OREQ;
939 expression_opt ::=
940                     // Expr
941         expression
942     |
943          ;
944 expression ::=
945                     // Expr
946         assignment_expression;
947 constant_expression ::=
948                     // Expr
949         expression;
```